Process Monitor HOW−TO for Linux

Table of Contents

Process Monitor HOW−TO for Linux.....1
Al Dev (Alavoor Vasudevan) alavoor@yahoo.com.....1
1. Linux or Unix Processes.....1
2. Unix/Linux command − procautostart.....1
3. File procautostart.cpp.....1
4. File debug.cpp.....1
5. File debug.h.....1
6. Makefile.....1
7. Testing the program − monitor_test.....1
8. Other Monitoring Tools.....1
9. Other Formats of this Document.....1
10. Copyright Notice.....1
1. Linux or Unix Processes.....1
2. Unix/Linux command − procautostart.....2
3. File procautostart.cpp.....3
4. File debug.cpp.....11
5. File debug.h.....12
6. Makefile.....12
7. Testing the program − monitor_test.....13
8. Other Monitoring Tools.....14
8.1 OpenSource Monitoring Tools.....14
8.2 Monitoring Tool − "daemontools".....14
8.3 Commercial Monitoring Tools.....15
9. Other Formats of this Document.....15
10. Copyright Notice.....17

Process Monitor HOW−TO for Linux

Al Dev (Alavoor Vasudevan) alavoor@yahoo.com

v8.0, 07 Mar 2001

This document describes how to monitor Linux/Unix processes and to re−start them automatically if they die without any manual intervention. This document also has URLs for "Unix Processes" FAQs.

---

1. Linux or Unix Processes

2. Unix/Linux command − procautostart

3. File procautostart.cpp

4. File debug.cpp

5. File debug.h

6. Makefile

7. Testing the program − monitor_test

8. Other Monitoring Tools

- 8.1 OpenSource Monitoring Tools
- 8.2 Monitoring Tool − "daemontools"
- 8.3 Commercial Monitoring Tools

9. Other Formats of this Document

10. Copyright Notice

---

1. Linux or Unix Processes

Processes are the "heart" of the Linux/Unix processes. It is very important to monitor the application processes to ensure 100% availability and reliability of the computer system. For example, processes of databases, web−server etc.. need to be up and running 24 hours a day and 365 days a year. Use the tools described in this document to the monitor important application processes.

See also the following related topics on Linux/Unix processes.

- Unix Programming FAQ − Chapter 1 Unix Processes http://www.erlenstar.demon.co.uk/unix/faq_toc.html
- Other FAQs on Unix are at http://www.erlenstar.demon.co.uk/unix/

2. Unix/Linux command − procautostart

Use the program **procautostart** (say "Prok−Auto−Start" or Process AutoStart) to monitor and automatically re−start any Unix/Linux process if they die. This tiny program is very powerful and is comparable to big commercial products which **costs about $80,000US**. Procautostart can be used for controlling following applications:

- For real−time control of process industries like chemical, manufacturing, power generation and others. Use *nano−seconds* in program to get fine control.
- For controlling processes of software applications like Web servers, database servers, mission critical unix processes, etc..
- As an alarm system for any general monitoring software system. The program can fire a pager or call cell phone or flash red lights on the computer screen. For calling a phone line you may need to use a Telephone card on PCI slot of the computer.

The program listing is given in following sections in this document.

procautostart −n < *delay_seconds* > **−c** "< *command_line* >" nohup &

This starts the unix process **procautostart** and also **command_line** process. The **procautostart** process will re−start **command_line** process if it dies. The *−n* option is the time delay in seconds before **procautostart** checks the running process started by **command_line**. It is advisable to start the procautostart as background process with no−hangup using "nohup &". See 'man nohup'.

The procautostart is written in "C" so that it is very fast and efficient, since the program is called every *n* seconds. Amount of resources consumed by procautostart is **very minute** and is negligible since the program size is small and is highly optimized with −o3 compiler option.

For example −

```
        procautostart −n 12 −c "monitor_test −d $HOME  −a dummy_arg " nohup &
```

Here **procautostart** will be checking the process monitor_test **every** 12 seconds.

The program will output log files in 'mon' sub−directory which has datetime stamp of when the processes died and re−started. These files gives info on how often the processes are dying.

You can also use micro−seconds option '−m' or nano−seconds option '−o', edit the source code file **procautostart.cpp** and uncomment appropriate lines.

3. File procautostart.cpp

// From your browser save this file as **text−file** named as 'procautostart.cpp'.

```
//
// Program to monitor the unix processes
// and automatically re-start them if they die
//

#include <stdio.h>
#include <strings.h> // C strings
#include <unistd.h> // for getopt
#include <alloc.h> // for free

#include <errno.h> // for kill() - error numbers command
extern int errno;

#ifdef Linux
#include <asm/errno.h> // for kill() - error numbers command
#endif

#include <sys/types.h> // for kill() command
#include <signal.h> // for kill() command
#include <sys/wait.h> // for wait()
#include <stdlib.h> // for setenv()
#include <time.h> // for strftime()
#include <libgen.h> // for basename()

#include "debug.h"

#define BUFF_HUN        100
#define BUFF_THOU       1024
#define PR_INIT_VAL     -10
#define WAIT_FOR_SYS    5  // wait for process to start up
#define DEF_SL_SECS     6  // default sleep time
#define SAFE_MEM        10  // to avoid any possible memory leaks

#define LOG_NO          false  // do not output to logfile
#define LOG_YES         true  // do output to logfile
#define STD_ERR_NO      false  // do not print to std err
#define STD_ERR_YES     true  // do print to std err
#define DATE_NO         false  // do not print date
#define DATE_YES        true  // do print date

int start_process(char *commandline, char *args[], char **envp, pid_t proc_pid);
int fork2(pid_t parent_pid, unsigned long tsecs);
inline void error_msg(char *mesg_out, char *lg_file, bool pr_lg, bool std_err, bool pr_dt);

//////////////////////////////////////////////
// To test this program use --
// procautostart -n 5 -c 'monitor_test dummy1 -a dummy2 -b dummy3  ' &
//////////////////////////////////////////////
int main(int argc, char **argv, char **envp)
{
        unsigned long   sleep_sec, sleep_micro, sleep_nano;
        int     ch;
        pid_t   proc_pid;
        int pr_no = PR_INIT_VAL;
        char mon_log[40];
```

```
char *pr_name = NULL, *cmd_line = NULL, **cmdargs = NULL;

// you can turn on debug by editing Makefile and put −DDEBUG in gcc
debug_("test debug", "this line");
debug_("argc", argc);

// Use getpid() − man 2 getpid()
proc_pid = getpid(); // get the Process ID of procautostart
debug_("PID proc_pid", (int) proc_pid);

// Create directory to hold log, temp files
system("mkdir mon 1>/dev/null 2>/dev/null");

sleep_sec = DEF_SL_SECS ; // default sleep time
sleep_micro = 0; // default micro−sleep time
sleep_nano = 0; // default nano−sleep time
optarg = cmd_line = NULL;
while ((ch = getopt(argc, argv, "n:m:o:h:c:")) != −1) // needs trailing colon :
{
        switch (ch)
        {
                case 'n':
                        debug_("scanned option n ", optarg);
                        sleep_sec = atoi(optarg);
                        debug_("sleep_sec", sleep_sec);
                        break;
                case 'm':
                        debug_("scanned option m ", optarg);
                        sleep_micro = atoi(optarg);
                        debug_("sleep_micro", sleep_micro);
                        break;
                case 'o':
                        debug_("scanned option o ", optarg);
                        sleep_nano = atoi(optarg);
                        debug_("sleep_nano", sleep_nano);
                        break;
                case 'c':
                        debug_("scanned option c ", optarg);
                        cmd_line = strdup(optarg); // does auto−malloc here
                        debug_("cmd_line", cmd_line);
                        break;
                case 'h':
                        debug_("scanned option h ", optarg);
                        fprintf(stderr, "\nUsage : %s −n <sleep> −m <microsecond> −o <nanosecond> −c '<command>'\n", argv[0]);
                        exit(−1);
                        break;
                default:
                        debug_("ch", "default");
                        fprintf(stderr, "\nUsage : %s −n <sleep> −m <microsecond> −o <nanosecond> −c '<command>'\n", argv[0]);
                        exit(−1);
                        break;
        }
}

if (cmd_line == NULL)
{
        fprintf(stderr, "\ncmd_line is NULL");
        fprintf(stderr, "\nUsage : %s −n <sleep> −m <microsecond> −o <nanosecond> −c '<command>'\n", argv[0]);
        exit(−1);
}
else
{
```

```
        // trim the trailing blanks -- otherwise problem in grep command
        int tmpii = strlen(cmd_line);
        for (int tmpjj = tmpii; tmpjj > -1; tmpjj--)
        {
                if (cmd_line[tmpjj] == ' ')
                        cmd_line[tmpjj] = '\0';
                else
                if (cmd_line[tmpjj] == '&')  // discards amp-and .. we will be appending later
                        cmd_line[tmpjj] = '\0';
                else
                if (cmd_line[tmpjj] == '\0')
                        continue;
                else
                {
                        if (cmd_line[tmpjj] == '&') // Discard trailing & in command line
                                cmd_line[tmpjj] = '\0';
                        break;
                }
        }
        debug_("cmd_line", cmd_line);
}

//argv0 = (char *) strdup(argv[0]);
//debug_("argv0", argv0);

// Start the process
{
        // Find the command line args
        char *aa = strdup(cmd_line), *bb = NULL;
        cmdargs = (char **) (malloc(sizeof(char **) + SAFE_MEM));
        for (int tmpii = 0; ; tmpii++)
        {
                // Allocate more memory ....
                cmdargs = (char **) realloc(cmdargs, (sizeof(char **) * (tmpii+1) + SAFE_MEM)  );

                if (tmpii == 0)
                        bb = strtok(aa, " ");
                else
                        bb = strtok(NULL, " "); // subsequent calls must have NULL as first arg

                if (bb == NULL)
                {
                        cmdargs[tmpii] = bb;
                        break;
                }
                else
                {
                        // Must malloc with strdup because aa, bb are
                        // local vars in local scope!!
                        cmdargs[tmpii] = strdup(bb);
                }

                debug_("tmpii", tmpii);
                debug_("cmdargs[tmpii]", (char *) cmdargs[tmpii]);
        }

        // In case execve you MUST NOT have trailing ampersand & in the command line!!
        //pr_no = start_process(cmd_line, NULL, NULL, proc_pid);  // Using execlp ...
        pr_no = start_process(cmdargs[0], & cmdargs[0], envp, proc_pid); // Using execve ....

        debug_("The child pid", pr_no);
        if (pr_no < 0)
```

```
        {
                fprintf(stderr, "\nFatal Error: Failed to start the process\n");
                exit(-1);
        }
        sleep(WAIT_FOR_SYS); // wait for the process to come up

        // Get process name - only the first word from cmd_line
        pr_name = strdup(basename(cmdargs[0])); // process name,  does auto-malloc here
}

// generate log file names
{
        char    aa[21];

        strncpy(aa, pr_name, 20); aa[20] = '\0';
        // Define mon file-names - make it unique with combination of
        // process name and process id
        sprintf(mon_log, "mon/%s%d.log", aa, (int) proc_pid);
}

// Print out pid to log file
if (pr_no > 0)
{
        char aa[200];
        sprintf(aa, "Process ID of %s is %d", pr_name, pr_no);
        error_msg(aa, mon_log, LOG_YES, STD_ERR_NO, DATE_YES);
}

// monitors the process - restarts if process dies...
bool process_died = false;
char print_log[200];
while (1)  // infinite loop - monitor every 6 seconds
{
        //debug_("Monitoring the process now...", ".");
        if (kill(pr_no, 0))  // if (kill(pr_no,0) != 0)
        {
                debug_("errno from kill() function", errno);
                if (errno == EINVAL)
                {
                        process_died = false; // unable to execute kill() - wrong input
                        strcpy(print_log, "Error EINVAL: Invalid signal was specified");
                        error_msg(print_log, mon_log, LOG_YES, STD_ERR_YES, DATE_YES);
                }
                else
                if (errno == ESRCH )
                {
                        // ERSRCH means - No process can be found corresponding to pr_no
                        // hence process had died !!
                        process_died = true; // No process can be found matching pr_no
                        sprintf(print_log,
                        "Error ESRCH: No process or process group can be found for %d", pr_no);
                        error_msg(print_log, mon_log, LOG_YES, STD_ERR_YES, DATE_YES);
                }
                else
                if (errno == EPERM)
                {
                        process_died = false; // unable to execute kill() - wrong input
                        strcpy(print_log,
                        "Error EPERM: The real or saved user ID does not match the real user ID");
                        error_msg(print_log, mon_log, LOG_YES, STD_ERR_YES, DATE_YES);
                }
                else
```

```
        {
                process_died = true; // process died!! restart now
                debug_("process_die ", "others");
        }

        if (process_died == true)
        {
                //
                // char respawn[1024];
                // strcpy(respawn, cmd_line);
                //
                // For "C" program use kill(pid_t process, int signal) function.
                // #include <signal.h> // See 'man 2 kill'
                // Returns 0 on success and -1 with errno set.
                //              kill -0 $pid 2>/dev/null || respawn
                // To get the exit return status do --
                //              kill -0 $pid 2>/dev/null | echo $?
                // Return value 0 is success and others mean failure
                // Sending 0 does not do anything to target process, but it tests
                // whether the process exists. The kill command will set its exit
                // status based on this process.
                //
                // Alternatively, you can use
                //              ps -p $pid >/dev/null 2>&1 || respawn
                // To get the exit return status do --
                //              ps -p $pid >/dev/null 2>&1 | echo $?
                // Return value 0 is success and others mean failure
                //

                // If the process had died, restart and  re-assign the pid to pr_no
                // start the process in background ....
                // Now re-assign new value of process id to pr_no
                if (pr_no > 0 )
                        sprintf(print_log, "Fatal Error: Process %s with PID = %d died!!",
                                pr_name, pr_no);
                else
                        sprintf(print_log, "Fatal Error: Process %s is not up!!",
                                pr_name);
                error_msg(print_log, mon_log, LOG_YES, STD_ERR_YES, DATE_YES);

                sprintf(print_log, "Starting process %s", pr_name);
                error_msg(print_log, mon_log, LOG_YES, STD_ERR_NO, DATE_NO);

                //pr_no = start_process(cmd_line, NULL, NULL, proc_pid);  // Using execlp ....
                pr_no = start_process(cmdargs[0], & cmdargs[0], envp, proc_pid); // Using execve ....

                debug_("The child pid", pr_no);
                if (pr_no < 0)
                {
                        sprintf(print_log, "Fatal Error: Failed to start the process");
                        error_msg(print_log, mon_log, LOG_YES, STD_ERR_YES, DATE_YES);
                        exit(-1);
                }
                sleep(WAIT_FOR_SYS); // wait for the process to come up
                sprintf(print_log, "Process ID of %s is %d", pr_name, pr_no);
                error_msg(print_log, mon_log, LOG_YES, STD_ERR_NO, DATE_NO);
        }
}
//debug_("Sleeping now ......", ".");
sleep(sleep_sec);

// Uncomment these to use micro-seconds
```

```
                // For real-time process control use micro-seconds or nana-seconds sleep functions
                // See 'man3 usleep', 'man 2 nanasleep'
                // If you do not have usleep() or nanosleep() on your system, use select() or poll()
                // specifying no file descriptors to test.
                //usleep(sleep_micro);

                // To sleep nano-seconds ...  Uncomment these to use nano-seconds
                //struct timespec *req = new struct timespec;
                //req->tv_sec = 0; // seconds
                //req->tv_nsec = sleep_nano; // nanoseconds
                //nanosleep( (const struct timespec *)req, NULL);
        }
}

inline void error_msg(char *mesg_out, char *lg_file, bool pr_lg, bool std_err, bool pr_dt)
{
        if (pr_lg) // (pr_lg == true) output to log file
        {
                char tmp_msg[BUFF_THOU];
                if (pr_dt == true) // print date and message to log file 'lg_file'
                {
                        sprintf(tmp_msg, "date >> %s; echo '\n%s\n' >> %s\n ",
                                lg_file, mesg_out, lg_file);
                        system(tmp_msg);
                }
                else
                {
                        sprintf(tmp_msg, "echo '\n%s\n' >> %s\n ",
                                mesg_out, lg_file);
                        system(tmp_msg);
                }
        }

        if (std_err) // (std_err == true) output to standard error
                fprintf(stderr, "\n%s\n", mesg_out);

        debug_("mesg_out", mesg_out);
}

// start a process and returns PID or -ve value if error
// The main() function has envp arg as in - main(int argc, char *argv[], char **envp)
int start_process(char *commandline, char *args[], char **envp, pid_t parent_pid)
{
        int ff;
        unsigned long  tsecs;

        tsecs = time(NULL); // time in secs since Epoch 1 Jan 1970
        debug_("Time tsecs", tsecs);

        // Use fork2() instead of fork to avoid zombie child processes
        switch (ff = fork2(parent_pid, tsecs))  // fork creates 2 process each executing the following lines
        {
        case -1:
                fprintf(stderr, "\nFatal Error: start_process() - Unable to fork process\n");
                _exit(errno);
                break;
        case 0:  // child process
                debug_("\nStarting the start child process\n", " ");
                // For child process to ignore the interrupts (i.e. to put
                // child process in "background" mode.
                // Signals are sent to all processes started from a
                // particular terminal. Accordingly, when a program is to be run non-interactively
```

```
// (started by &), the shell arranges that the program will ignore interrupts, so
// it won't be stopped by interrupts intended for foreground processes.
// Hence if previous value of signal is not IGN than set it to IGN.

// Note: Signal handlers cannot be set for SIGKILL, SIGSTOP
if (signal(SIGINT, SIG_IGN) == SIG_ERR)
        fprintf(stderr, "\nSignal Error: Not able to set signal to SIGINT\n");
else
if (signal(SIGINT, SIG_IGN) != SIG_IGN) // program already run in background
        signal(SIGINT, SIG_IGN);  // ignore interrupts

if (signal(SIGHUP, SIG_IGN) == SIG_ERR)
        fprintf(stderr, "\nSignal Error: Not able to set signal to SIGHUP\n");
else
if (signal(SIGHUP, SIG_IGN) != SIG_IGN) // program already run in background
        signal(SIGHUP, SIG_IGN);  // ignore hangups

if (signal(SIGQUIT, SIG_IGN) == SIG_ERR)
        fprintf(stderr, "\nSignal Error: Not able to set signal to SIGQUIT\n");
else
if (signal(SIGQUIT, SIG_IGN) != SIG_IGN) // program already run in background
        signal(SIGQUIT, SIG_IGN);  // ignore Quit

if (signal(SIGABRT, SIG_IGN) == SIG_ERR)
        fprintf(stderr, "\nSignal Error: Not able to set signal to SIGABRT\n");
else
if (signal(SIGABRT, SIG_IGN) != SIG_IGN) // program already run in background
        signal(SIGABRT, SIG_IGN);  // ignore ABRT

if (signal(SIGTERM, SIG_IGN) == SIG_ERR)
        fprintf(stderr, "\nSignal Error: Not able to set signal to SIGTERM\n");
else
if (signal(SIGTERM, SIG_IGN) != SIG_IGN) // program already run in background
        signal(SIGTERM, SIG_IGN);  // ignore TERM

// sigtstp − Stop typed at tty. Ignore this so that parent process
// be put in background with CTRL+Z or with SIGSTOP
if (signal(SIGTSTP, SIG_IGN) == SIG_ERR)
        fprintf(stderr, "\nSignal Error: Not able to set signal to SIGTSTP\n");
else
if (signal(SIGTSTP, SIG_IGN) != SIG_IGN) // program already run in background
        signal(SIGTSTP, SIG_IGN);  // ignore TSTP

// You can use debug_ generously because they do NOT increase program size!
debug_("before execve commandline", commandline);
debug_("before execve args[0]", args[0]);
debug_("before execve args[1]", args[1]);
debug_("before execve args[2]", args[2]);
debug_("before execve args[3]", args[3]);
debug_("before execve args[4]", args[4]);
debug_("before execve args[5]", args[5]);
debug_("before execve args[6]", args[6]);
debug_("before execve args[7]", args[7]);
execve(commandline, args, envp);

// execlp, execvp does not provide expansion of metacharacters
// like <, >, *, quotes, etc., in argument list. Invoke
// the shell /bin/sh which then does all the work. Construct
// a string 'commandline' that contains the complete command
//execlp("/bin/sh", "sh", "−c", commandline, (char *) 0);  // if success than NEVER returns !!

// If execlp returns than there is some serious error !! And
```

```
                // executes the following lines below...
                fprintf(stderr, "\nFatal Error: Unable to start child process\n");
                ff = -2;
                exit(127);
                break;
        default: // parent process
                // child pid is ff;
                if (ff < 0)
                        fprintf(stderr, "\nFatal Error: Problem while starting child process\n");

                {
                        char    buff[BUFF_HUN];
                        FILE    *fp1;
                        sprintf(buff, "mon/%d%lu.out", (int) parent_pid, tsecs); // tsecs is unsigned long
                        fp1 = fopen(buff, "r");
                        if (fp1 != NULL)
                        {
                                buff[0] = '\0';
                                fgets(buff, BUFF_HUN, fp1);
                                ff = atoi(buff);
                        }
                        fclose(fp1);
                        debug_("start process(): ff - ", ff);
#ifndef DEBUG
                        sprintf(buff, "rm -f mon/%d%lu.out", (int) parent_pid, tsecs);
                        system(buff);
#endif // DEBUG
                }

                // define wait() to put child process in foreground or else put in background
                //waitpid(ff, & status, WNOHANG || WUNTRACED);
                //waitpid(ff, & status, WUNTRACED);
                //wait(& status);

                break;
        }
        return ff;
}

/* fork2() -- like fork, but the new process is immediately orphaned
 *            (won't leave a zombie when it exits)
 * Returns 1 to the parent, not any meaningful pid.
 * The parent cannot wait() for the new process (it's unrelated).
 */
/* This version assumes that you *haven't* caught or ignored SIGCHLD. */
/* If you have, then you should just be using fork() instead anyway.  */

int fork2(pid_t parent_pid, unsigned long tsecs)
{
    pid_t mainpid, child_pid = -10;
    int status;
        char    buff[BUFF_HUN];

    if (!(mainpid = fork()))
    {
        switch (child_pid = fork())
        {
          case 0:
                        //child_pid = getpid();
                        //debug_("At case 0 fork2 child_pid : ", child_pid);
                        return 0;
          case -1:
```

```
                        _exit(errno);    /* assumes all errnos are <256 */
                default:
                        debug_("fork2 child_pid : ", (int) child_pid);
                        sprintf(buff, "echo %d > mon/%d%lu.out", (int) child_pid, (int) parent_pid, tsecs);
                        system(buff);
                        _exit(0);
            }
        }

            //debug_("fork2 pid : ", pid);
        if (mainpid < 0 || waitpid(mainpid, & status, 0) < 0)
          return -1;

        if (WIFEXITED(status))
          if (WEXITSTATUS(status) == 0)
            return 1;
          else
            errno = WEXITSTATUS(status);
        else
          errno = EINTR;  /* well, sort of :-) */

        return -1;
}
```

4. File debug.cpp

// From your browser save this file as **text−file** named as 'debug.cpp'.

```
// This file defines the debug_() function which can be used for debugging
// the program. It is similar to "C" assert(). The debug_() is set to void()
// if DEBUG is not defined in Makefile. This way executable size of production
// release is NOT AT ALL effected. Using debug_() very generously has no
// impact on production executable size.
#ifdef DEBUG

#include "debug.h"
// Variable value[] can be char, string, int, unsigned long, float, etc...

void local_dbg(char name[], char value[], char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %s\n", fname, lineno, name, value ); }

void local_dbg(char name[], int value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %d\n", fname, lineno, name, value ); }

void local_dbg(char name[], unsigned int value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %d\n", fname, lineno, name, value ); }

void local_dbg(char name[], long value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %d\n", fname, lineno, name, value ); }

void local_dbg(char name[], unsigned long value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %d\n", fname, lineno, name, value ); }

void local_dbg(char name[], short value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %d\n", fname, lineno, name, value ); }
```

```
void local_dbg(char name[], unsigned short value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %d\n", fname, lineno, name, value ); }

void local_dbg(char name[], float value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %f\n", fname, lineno, name, value ); }

void local_dbg(char name[], double value, char fname[], int lineno, bool logfile) {
        printf("\nDebug %s Line: %d %s is = %f\n", fname, lineno, name, value ); }

// You add many more here - value can be a class, ENUM, datetime, etc...

#endif // DEBUG
```

5. File debug.h

// From your browser save this file as **text−file** named as 'debug.h'.

```
#ifdef DEBUG

#include <stdio.h>
//#include <strings.h>
//#include <assert.h>  // assert() macro which is also used for debugging

// Debugging code
// Use debug2_ to output result to a log file
#define debug_(NM, VL) (void) ( local_dbg(NM, VL, __FILE__, __LINE__) )
#define debug2_(NM, VL, LOG_FILE) (void) ( local_dbg(NM, VL, __FILE__, __LINE__, LOG_FILE) )
void local_dbg(char name[], char value[], char fname[], int lineno, bool logfile= false);
void local_dbg(char name[], int value, char fname[], int lineno, bool logfile= false);
void local_dbg(char name[], unsigned long value, char fname[], int lineno, bool logfile= false);
void local_dbg(char name[], float value, char fname[], int lineno, bool logfile= false);

#else

#define debug_(NM, VL) ((void) 0)
#define debug2_(NM, VL, LOG_FILE) ((void) 0)

#endif // DEBUG
```

6. Makefile

From your browser save this file as **text−file** named as 'Makefile'.

```
EXE=procautostart
SRCS=procautostart.cpp debug.cpp
OBJS=procautostart.o  debug.o

CXX=gcc

HOSTFLAG=-DLinux
```

```
#HOSTFLAG=-DSunOS

# Do not use compiler optimizer -O as this may break the program
# Use debug flag to enable the debug() function. If DEBUG is not
# defined than the function debug() is set to void(), similar
# to assert()
# Use options -Wall (all warning msgs) -O3 (optimization)
#MYCFLAGS=-DDEBUG -g -Wall
MYCFLAGS=-O3 -Wall

all: $(OBJS)
        $(CXX) $(HOSTFLAG) $(MYCFLAGS) $(OBJS) -o $(EXE)

$(OBJS): $(SRCS)
        $(CXX) -c $(HOSTFLAG) $(MYCFLAGS) $(SRCS)

clean:
        rm -f *.o *.log *.log.old *.pid core err a.out afiedt.buf
        rm -f $(EXE)
```

7. Testing the program − monitor_test

From your browser save this file as **text−file** named as 'monitor_test'.

Use this program for testing the 'procautostart' program. For example −

```
        procautostart -n 12 -c "monitor_test -d $HOME  -a dummy_arg " nohup &
```

Here **procautostart** will be checking the process monitor_test **every** 12 seconds.

```
#!/bin/ksh

# Program to test the procautostart

echo "Started the monitor_test ...."
date > monitor_test.log
while :
do
        date >> monitor_test.log
        sleep 2
done
```

Then do a tail command to monitor the output. And simulate the failures of monitor_test programs.

```
        bash$ tail -f monitor_test.log
        bash$ ps -ef | grep monitor_test
See the PID of monitor_test and kill it..
        bash$ kill -9 < PID of monitor_test >
```

```
Once you kill the process, you will notice that it immediately comes alive due to procautostart !
```

8. Other Monitoring Tools

8.1 OpenSource Monitoring Tools

On linux systems you can find the following packages. If it is not in the main cdrom than you must check in the contrib cdrom :

- On contrib cdrom **daemontools*.rpm**
- 'top' command **procps*.rpm**
- 'top' command graphic mode **procps−X11*.rpm**
- 'ktop' graphic mode **ktop*.rpm**
- 'gtop' graphic mode **gtop*.rpm**
- 'WMMon' CPU load **wmmon*.rpm**
- 'wmsysmon' monitor **wmsysmon*.rpm**
- 'procmeter' System activity meter **procmeter*.rpm**

To use top commands type at unix prompt −

```
        $ top
        $ ktop
        $ gtop
```

8.2 Monitoring Tool − "daemontools"

Visit the web site of daemontools at http://www.pobox.com/~djb/daemontools.html

To install the daemontools RPM, do −

```
        # rpm −i /mnt/cdrom/daemontools*.html
        # man supervise
```

supervise monitors a service. It starts the service and restarts the service if it dies. The companion svc program stops, pauses, or restarts the service on sysadmin request. The svstat program prints a one−line status report. See man page by 'man supervise'

svc − control a supervised service.

svc changes the status of a supervise−monitored service. dir is the same directory used for supervise. You can list several dirs. svc will change the status of each service in turn.

svstat − print the status of a supervised service.

svstat prints the status of a supervise−monitored service. dir is the same directory used for supervise. You can list several dirs. svstat will print the status of each service in turn.

cyclog writes a log to disk. It automatically synchronizes the log every 100KB (by default) to guarantee data integrity after a crash. It automatically rotates the log to keep it below 1MB (by default). If the disk fills up, cyclog pauses and then tries again, without losing any data. See man page by 'man cyclog'

accustamp puts a precise timestamp on each line of input. The timestamp is a numeric TAI timestamp with microsecond precision. The companion tailocal program converts TAI timestamps to local time. See 'man accustamp'

usually watches a log for lines that do not match specified patterns, copying those lines to stderr. The companion errorsto program redirects stderr to a file. See 'man usually'

setuser runs a program under a user's uid and gid. Unlike su, setuser does not gain privileges; it does not check passwords, and it cannot be run except by root. See 'man setuser'

8.3 Commercial Monitoring Tools

There are commercial monitoring tools available. Check out −

- BMC Patrol for Unix/Databases http://www.bmc.com
- TIBCO corp's Hawk for Unix monitoring http://www.tibco.com
- LandMark corporation
- Platinum corporation

---

9. Other Formats of this Document

This document is published in 11 different formats namely − DVI, Postscript, Latex, Adobe Acrobat PDF, LyX, GNU−info, HTML, RTF(Rich Text Format), Plain−text, Unix man pages and SGML.

- You can get this HOWTO document as a single file tar ball in HTML, DVI, Postscript or SGML formats from − ftp://sunsite.unc.edu/pub/Linux/docs/HOWTO/other−formats/
- Plain text format is in: ftp://sunsite.unc.edu/pub/Linux/docs/HOWTO
- Translations to other languages like French, German, Spanish, Chinese, Japanese are in ftp://sunsite.unc.edu/pub/Linux/docs/HOWTO Any help from you to translate to other languages is welcome.

The document is written using a tool called "SGML−Tools" which can be got from − http://www.sgmltools.org Compiling the source you will get the following commands like

- sgml2html CVS−HOWTO.sgml (to generate html file)
- sgml2rtf CVS−HOWTO.sgml (to generate RTF file)
- sgml2latex CVS−HOWTO.sgml (to generate latex file)

LaTeX documents may be converted into PDF files simply by producing a Postscript output using **sgml2latex** (and dvips) and running the output through the Acrobat **distill** (http://www.adobe.com) command as follows:

```
bash$ man sgml2latex
bash$ sgml2latex filename.sgml
bash$ man dvips
bash$ dvips -o filename.ps filename.dvi
bash$ distill filename.ps
bash$ man ghostscript
bash$ man ps2pdf
bash$ ps2pdf input.ps output.pdf
bash$ acroread output.pdf &
```

Or you can use Ghostscript command **ps2pdf**. ps2pdf is a work−alike for nearly all the functionality of Adobe's Acrobat Distiller product: it converts PostScript files to Portable Document Format (PDF) files. **ps2pdf** is implemented as a very small command script (batch file) that invokes Ghostscript, selecting a special "output device" called **pdfwrite**. In order to use ps2pdf, the pdfwrite device must be included in the makefile when Ghostscript was compiled; see the documentation on building Ghostscript for details.

This howto document is located at −

- http://sunsite.unc.edu/LDP/HOWTO/CVS−HOWTO.html

Also you can find this document at the following mirrors sites −

- http://www.caldera.com/LDP/HOWTO/CVS−HOWTO.html
- http://www.WGS.com/LDP/HOWTO/CVS−HOWTO.html
- http://www.cc.gatech.edu/linux/LDP/HOWTO/CVS−HOWTO.html
- http://www.redhat.com/linux−info/ldp/HOWTO/CVS−HOWTO.html
- Other mirror sites near you (network−address−wise) can be found at http://sunsite.unc.edu/LDP/hmirrors.html select a site and go to directory /LDP/HOWTO/CVS−HOWTO.html

In order to view the document in dvi format, use the xdvi program. The xdvi program is located in tetex−xdvi*.rpm package in Redhat Linux which can be located through ControlPanel | Applications | Publishing | TeX menu buttons. To read dvi document give the command −

```
xdvi -geometry 80x90 howto.dvi
man xdvi
```

And resize the window with mouse. To navigate use Arrow keys, Page Up, Page Down keys, also you can use 'f', 'd', 'u', 'c', 'l', 'r', 'p', 'n' letter keys to move up, down, center, next page, previous page etc. To turn off expert menu press 'x'.

You can read postscript file using the program 'gv' (ghostview) or 'ghostscript'. The ghostscript program is in ghostscript*.rpm package and gv program is in gv*.rpm package in Redhat Linux which can be located through ControlPanel | Applications | Graphics menu buttons. The gv program is much more user friendly than ghostscript. Also ghostscript and gv are available on other platforms like OS/2, Windows 95 and NT, you view this document even on those platforms.

- Get ghostscript for Windows 95, OS/2, and for all OSes from http://www.cs.wisc.edu/~ghost

To read postscript document give the command −

```
gv howto.ps
ghostscript howto.ps
```

You can read HTML format document using Netscape Navigator, Microsoft Internet explorer, Redhat Baron Web browser or any of the 10 other web browsers.

You can read the latex, LyX output using LyX a X−Windows front end to latex.

---

10. Copyright Notice
