Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Системное программное обеспечение вычислительных машин

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«SLM — System Linux Monitor»

БГУИР КП 1-40 02 01 122* ПЗ

Студент: гр. 550501 Чистяков М. Ю.

Руководитель: Лавникевич Д.А.

Минск 2017

Содержание

ВВЕДЕНИЕ

Linux — общее название семейства Unix-подобных операционных систем на базе одного и того же одноименного ядра. Как правило, такие системы (как и само ядро) разрабатываются с помощью набора утилит и программ проекта GNU, поэтому часто ядро называют GNU/Linux. Как и само ядро, системы на его основе обычно создаются и распространяются в соответствии с моделью разработки свободного программного обеспечения. Распространяются эти системы в виде различных дистрибутивов — в форме, готовой для установки и удобной для сопровождения и обновлений, — и имеющих собственный набор приложений и утилит.

Само ядро GNU/Linux появилось в начале 1990-х годов. На сегодняшний день операционные системы на основе этого ядра занимают большую долю на рынке операционных систем для суперкомьютеров и серверов. Последнее время их применение расширяется на встраиваемые системы и мобильные устройства, а также на персональные компьютеры.

Низкоуровневый api ядра позволяет выполнить широкий спектр задач, однако для конечного пользователя он неприменим. Посредником между пользователем и ядром служат различные специализированные утилиты, которых существует большое количество. Их идею можно выразить словами «делай что-то одно, но делай это качественно». Зачастую утилиты имеют большое количество аргументов, позволяющих произвести точную настройку выполнения.

Большинство современных операционных систем имеют специальную утилиту, предназначенную исключительно для установки других утилит и их сопровождения в системе — пакетный менеджер. После того, как утилита будет упакована в пакет специального формата, процесс установки для конечного пользователя будет значительно упрощен.

Данная курсовая работа посвящена написанию программы-утилиты для системы на основе ядра Linux (Debian), предназначенной для отслеживания (мониторинга) системных событий, таких как операции с файлами, подключение питания или сетевого менеджера. Эта утилита может быть полезна системным администраторам для анализа состояния системы.

1. ПОСТАНОВКА ЗАДАЧИ

Результатом работы должен стать бинарный debian-пакет, готовый для установки в операционной системе пользователя. Пакет состоит из двух составляющих:

- утилита slm — позволяет отслеживать события в реальном времени, используя в качестве вывода стандартный поток stdout (по умолчанию связан с терминалом, из которого запущена утилита). Данная версия позволяет отслеживать только один класс событий. При получении сигнала SIGINT (ctrl-c) утилита должна корректно завершать свою работу.
- демон slmd — имеет аналогичный с утилитой функционал, но другую механику работы. Если утилита должна запускаться из командной строки, то демон должен быть доступен как сервис в другой стандартной утилите systemctl. Systemctl — это команда для работы с systemd, менеджером системы и служб Linux, под управлением которого должен находится и slmd. Таким образом, пользователь сможет управлять работой утилиты в фоновом режиме, не вдаваясь в подробности работы, такие как pid запущенного демона. Конфигурация демона должна производиться с помощью специального файла, расположенного в /etc/conf/slmd.conf; собранная демоном информация перенаправляется в файл /var/log/slmd.log. Важное отличие демона от утилиты — возможность мониторинга несколько событий одновременно.

События, на которые должна реагировать утилита:

- открытие, закрытие, чтение, удаление и переименование файла
- подключение и отключение питания
- подключение и отключение bluetooth
- статус работы network manager
- подключение и отключение внешних носителей памяти

2. ОБЗОР АНАЛОГОВ

slm реализована с помощью нескольких подсистем ядра Linux, которые имеют свои интерфейсные утилиты. Их можно выделить как непосредственные аналоги. slm, конечно, не может сравниться с ними в функциональности, но отличается простотой для конечного пользователя. Перечислим основные аналоги и базовые методы работы с ними.

2.1 D-feet

Для работы с network manager и отслеживания событий подключения внешних накопителей используется подсистема Dbus. Dbus — средство межпроцессного взаимодействия, активно используемая современными графическими оболочками (GNOME и KDE) и другими приложениями. D-feet предоставляет графический интерфейс для просмотра подключенных к шине Dbus объектов, вызова некоторых их методов, просмотра значений свойств и списка доступных сигналов. Программа не является стандартной и для её установки необходимо воспользоваться командой `sudo apt-get install d-feet`. Ниже приведен скриншот работы программы, на котором автор получил скорость вращения своего жесткого диска.

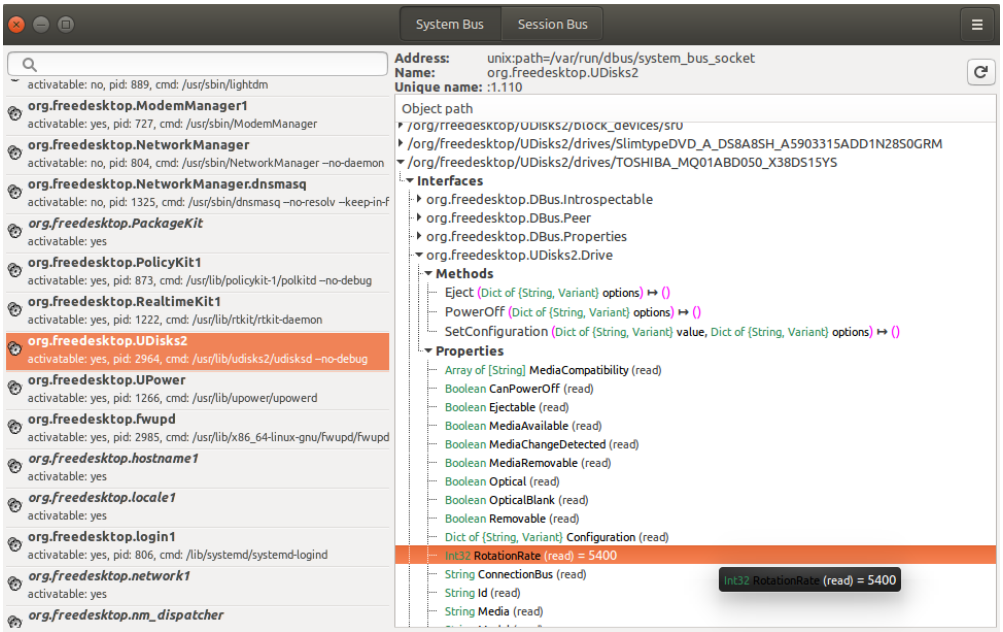

Рис. 2.1.1 — D-feet

2.2 udevadm

udevadm — интерфейсная утилита для подсистемы udev. Udev предназначена для отслеживания текущего состояния подключенного оборудования, именно udev отвечает за наличие актуальных данных в каталоге /dev. Udev работает в режиме пользователя, получая данные из ядра с помощью системного сокета, а затем фильтрует их, используя специальные правила (которые можно изменять пользователю в своих целях). Утилита позволяет увидеть как события, полученные непосредственно из ядра (KERNEL), так и события после применения udev-правил (UDEV). Пример вывода утилиты (урезан) при подключении флешки представлен в листинге ниже.

```
zoxal@zoxal-ubu:~$ udevadm monitor
monitor will print the received events for:
UDEV - the event which udev sends out after rule processing
KERNEL - the kernel uevent

KERNEL[11307.575960] add /devices/pci0000:00/0000:00:12.2/usb1/1-5 (usb)
KERNEL[11307.578158] add /devices/pci0000:00/0000:00:12.2/usb1/1-5/1-5:1.0 (usb)
UDEV  [11307.776217] add /devices/pci0000:00/0000:00:12.2/usb1/1-5 (usb)
KERNEL[11308.142140] add /module/usb_storage (module)
KERNEL[11308.144587] add /devices/pci0000:00/0000:00:12.2/usb1/1-5/1-5:1.0/host6 (scsi)
KERNEL[11308.144623] add /devices/pci0000:00/0000:00:12.2/usb1/1-5/1-5:1.0/host6/scsi_host/host6 (scsi_host)
UDEV  [11308.144639] add /module/usb_storage (module)
KERNEL[11308.144649] add /bus/usb/drivers/usb-storage (drivers)
UDEV  [11308.145624] add /bus/usb/drivers/usb-storage (drivers)
KERNEL[11308.176667] add /module/uas (module)
KERNEL[11308.176770] add /bus/usb/drivers/uas (drivers)
UDEV  [11308.178414] add /devices/pci0000:00/0000:00:12.2/usb1/1-5/1-5:1.0 (usb)
```

2.3 inotifywait

inotifywait — интерфейсная утилита для подсистемы inotify. inotify позволяет получать уведомления о событиях, связанных с файлами и каталогами файловой системы. Например, открытие файлов и каталогов для чтения или записи, изменения атрибутов, перемещение, удаление. В основное ядро была включена начиная с 2.6.13, но интерфейсная утилита туда не входит, поэтому для работы необходимо установить соответствующий пакет командой `sudo apt-get install inotify-tools`. Ниже представлен урезанный вывод утилиты при открытии домашней папки пользователя и создании там одного файла.

```
zoxal@zoxal-ubu:~$ inotifywait ~ -m
Setting up watches.
Watches established.
/home/zoxal/ OPEN,ISDIR
/home/zoxal/ ACCESS,ISDIR
/home/zoxal/ ACCESS,ISDIR
/home/zoxal/ CLOSE_NOWRITE,CLOSE,ISDIR
/home/zoxal/ CREATE inotify.test
/home/zoxal/ OPEN inotify.test
/home/zoxal/ ATTRIB inotify.test
/home/zoxal/ CLOSE_WRITE,CLOSE inotify.test
```

Можно заметить, что вывод стандартных утилит при отсутствии специальных настроек достаточно многословен. Одна из задач, которая ставилась при реализации slm, заключалась в лаконичности записей журнала. Пример вывода работы slmd (файл /var/log/slmd.log) приведен ниже.

```
Thu Jun  1 18:26:45 2017 [INFO]: file /home/zoxal/getopt_test.c was closed
Thu Jun  1 18:27:00 2017 [INFO]: power supply on
Thu Jun  1 18:27:03 2017 [INFO]: power supply off
Thu Jun  1 18:28:03 2017 [INFO]: Disk 'Silicon-Power8G' has been connected via 'usb'
```

3 ОБЗОР ИСПОЛЬЗУЕМЫХ ИНСТРУМЕНТОВ И БИБЛИОТЕК

Самый главный инструмент в программировании — это, конечно, сам язык программирования. slm написан исключительно на языке C. Один из главных агрументов в пользу использования языка C для реализации slm состоит в том, что само ядро Linux написано преимущественно на C, соответственно api подсистем могут быть использованы напрямую.

В качестве среды разработки используется CLion (JetBrains).

3.1 CMake

CMake — утилита сборки проектов на C и C++. Главная особенность в том, что CMake производит сборку непросредственно. CMake детектирует присутствующие в системе утилиты сборки и генерирует их сборочные скрипты из своих скриптов. Таким образом достигается кроссплатформенность проекта — его можно будет собрать в любой системе, где присутствует известная для CMake утилита сборки, которых на сегодняшний день очень много. Язык скриптов CMake имеет богатый синтаксис, позволяет использовать условные конструкции, циклы и даже функции. Скрипты CMake как правило находятся в файле CMakeLists.txt, которых может быть несколько в случае многомодульного приложения. Ниже приведена часть скрипта CmakeLists.txt для сборки самой утилиты slm:

```
set(MONITOR_SRC src/monitors/monitor.c src/monitors/inotify_monitor.c
    src/monitors/dbus_monitor.c src/monitors/udev_monitor.c
    )

add_executable(slm src/utility/main.c ${MONITOR_SRC} src/logging.c)
set(THREADS_PREFER_PTHREAD_FLAG ON)
find_package (Threads)
target_link_libraries (slm ${CMAKE_THREAD_LIBS_INIT}
    ${GLIB2_LIBRARIES}
    ${GIO2_LIBRARIES}
```

```
${UDEV_LIBRARIES})
```

Также существует много дополнительный модулей, которые расширяют функционал CMake. Например, в данном курсовом проекте для сборки бинарного deb-пакета использовался модуль CPack. CPack позволяет собирать приложение в различные дополнительные форматы помимо исполняемых файлов: архивы (zip, tar), пакеты (rpm, deb) и другие. Его использование значительно упрощает задачу упаковки приложения для стандартных случаев. Ниже для демонстрации приведена часть конфигурации CPack, используемая в курсовом проекте.

```
SET(CPACK_DEBIAN_PACKAGE_MAINTAINER "zzoxall@gmail.com")
  SET(CPACK_DEBIAN_PACKAGE_DESCRIPTION "system linux monitor")
  SET(CPACK_DEBIAN_PACKAGE_DESCRIPTION_SUMMARY
      "slm utitity and slmd daemon aimed to monitor linux system events")
```

3.2 GLib

GLib — библиотека, расширяющая возможности, предоставляемые стандартной библиотекой libc языка C. Разрабатывается в рамках проектов GTK+ и GNOME, но применяется далеко не только в графических приложениях. На сегодняшний день это одна из самых мощных по функционалу библиотек C. И, что важно в рамках данного курсового проекта, GLib предоставляет качественную обертку над низкоуровневым интерфейсом DBus. Именно с помощью GLib происходит взаимодействие slm с подсистемой DBus.

Ещё одна часть GLib, используемая в проекте — динамические массивы GArray. В таком массиве удобно хранить запущенные в демоне slmd мониторы, так как заранее количество мониторов, указанных в файле конфигурации, неизвестно. Ниже приведён пример использования GArray в проекте.

```
if (monitor_from_args(argc, argv, &new_monitor) == CALL_SUCCESS) {
        g_array_append_val(monitors_array, new_monitor);
        monitors_array_size++;
        start_monitor(new_monitor);
```

```
}
…
for (int i = 0; i < monitors_array_size; i++) {
        monitor_t monitor = g_array_index(monitors_array, monitor_t, i);
        stop_monitor(monitor);
}
for (int i = 0; i < monitors_array_size; i++) {
        monitor_t monitor = g_array_index(monitors_array, monitor_t, i);
        join_monitor(monitor);
        destroy_monitor(monitor);
}
```

3.3 PThreads

PThreads (POSIX thread) — стандарт POSIX по реализации потоков выполнения. Linux, конечно имеет свою одноименную библиотеку, реализующую данный стандарт. slm спроектирована таким образом, что каждый отдельный монитор запускается в отдельном потоке. Такое разделение позволяет мониторам работать асинхронно, не блокируя работу друг друга. Также в проекте используется мьютексы для синхронизации доступа ко флагу состояния монитора. Пример использования библиотеки приведен ниже.

```
int udev_stop(monitor_t monitor) {
        pthread_mutex_lock(&(monitor->udev->state_mutex));
        if (monitor->state != MONITOR_STATE_RUNNING) {
                pthread_mutex_unlock(&(monitor->udev->state_mutex));
                return E_MONITOR_INVALID_STATE;
        }
        monitor->state = MONITOR_STATE_DYING;
        pthread_mutex_unlock(&(monitor->udev->state_mutex));
        return CALL_SUCCESS;
}
```

3.4 getopt

getopt и getopt_long — функции, созданные в проекте GNU для парсинга аргументов командной строки. В данном проекте использовались для парсинга типов событий, на которые должен быть подписан inotify-монитор. Пример использования в курсовом проекте приведен ниже.

```
while ((c = getopt(argc, argv, argument_string)) != -1) {
        switch (c) {
                case MODE_OPEN: {
                        inotify_monitor->mode[strlen(inotify_monitor->mode)] = MODE_OPEN;
                        break;
                }
                case MODE_WRITE: {
                        inotify_monitor->mode[strlen(inotify_monitor->mode)] = MODE_WRITE;
                        break;
                }
                case MODE_CLOSE: {
                        inotify_monitor->mode[strlen(inotify_monitor->mode)] = MODE_CLOSE;
                        break;
                }
                case MODE_MOVE: {
                        inotify_monitor->mode[strlen(inotify_monitor->mode)] = MODE_MOVE;
                        break;
                }
                case MODE_DELETE: {
                        inotify_monitor->mode[strlen(inotify_monitor->mode)] = MODE_DELETE;
                        break;
                }
                case '?':
                default: {
```

```
                        free(inotify_monitor->mode);
                        free(inotify_monitor);
                        free((*monitor));
                        return E_INVALID_MONITOR_ARGUMENT;
                    }
                }
            }
```

3.5 poll и epoll

poll и epoll — системные функции, которые позволяют выполнить timeout-limited ожидание готовности одного из перечисленных в агрументах файловых дескрипторорах для выполнения желаемой операции, например, POLLIN для чтения или POLLOUT для записи. Использование этих функций позволило избежать блокировки при чтении файловых дескрипторов событий inotify и udev для возможности корректного завершения потока монитора. Пример использования функции poll в проекте приведен ниже.

```
do {
        pthread_mutex_lock(&(inotify_monitor->state_mutex));
        if (monitor->state == MONITOR_STATE_DYING) {
                pthread_mutex_unlock(&(inotify_monitor->state_mutex));
                mark_dead(monitor);
                return NULL;
        }
        pthread_mutex_unlock(&(inotify_monitor->state_mutex));
   poll(&inotify_poll_fd, 1, 500);
} while (inotify_poll_fd.revents != POLLIN);
```

4 АРХИТЕКТУРА ПРИЛОЖЕНИЯ

4.1 Общий обзор

Каждая задача по отслеживанию системного события заданного типа выполняется монитором. Мониторы в приложении классифицированы по типу механизма, который они используют для получения событий. Созданы мониторы intotify_monitor, dbus_monitor и udev_monitor. Каждый монитор

должен реализовать свой простой интерфейс по инициализации, запуску, остановке и разрушению. Такая архитектура была выбрана для возможности последующего расширения приложения новыми модулями. Основная структура монитора и требуемые интерфейсы представлены в заголовочном файле monitor.h, основное содержание которого представлено ниже.

```
struct monitor_t {
	int type;
	union {
		inotify_monitor_t inotify;
		dbus_monitor_t dbus;
		udev_monitor_t udev;
	};
	int state;
};
typedef struct monitor_t* monitor_t;

int monitor_from_args(int argc, char* argv[], monitor_t*);
int start_monitor(monitor_t);
int stop_monitor(monitor_t);
void join_monitor(monitor_t);
int destroy_monitor(monitor_t);
```

monitor_t — обобщённый интерфейс любого slm-монитора. Конкретные реализации мониторов и их структуры будут рассмотрены ниже. Важно, что любой монитор должен отписаться от прослушивания всех сигналов, если он выполняется в отдельном потоке.

Приложение изначально проектировалось таким образом, чтобы можно было легко создать два независимых исполняемых модуля (утилиту slm и демон slmd), максимально используя один и тот же код для реализации основного функционала. Для этого часть приложения, отвечающая непосредственно за работу мониторов, была выделена в статическую библиотеку, главным файлом которой является monitor.h (рассмотрен выше).

В отдельный интерфейс также выделено логирование.

log_info("file %s was opened", inotify_monitor->file_path);

Для того, чтобы обеспечить безопасный вывод сообщений от разных мониторов, использовался мьютекс.

4.2 Работа демона slmd

Демоном в Linux называется процесс, который работает в фоновом режиме без прямого взаимодействия с пользователем и не подключен ни к какому терминалу. Разработка демона — ответственная и непростая задача. Так как демона сложно контролировать, он должен работать безотказно и обрабатывать свои ошибки без вреда для операционной системы. Независимость демона также чрезвычайно усложняет процесс отладки (если вообще можно говорить об отладке демона), в чем автор убедился самостоятельно.

Утилита slmd в начале работы превращается в демона. Для этого выполняется следующая последовательность действий:

1. fork(). Демон должен в работать в независимом процессе. Для того, чтобы избавиться от родительского процесса, родительский процесс вызывает fork и завершает свою работу.
2. umask(0). Демон наследует права по умолчанию от родительского процесса. Лучше всего их сбросить, чтобы не было сюрпризов в дальнейшем
3. setsid(). Данный вызов создаёт новую сессию и переводит вызывающий процесс в неё. Это позволяет сделать демона независимым от сессии пользователя, который вызвал инициализацию демона. Также важно, что этот вызов разорвет связь ребёнок-родитель между процессом демоном и вызывающим процессом, что не позволит демону стать зомби-процессом в дальнейшем.
4. chdir(«/»). Так же, как и с правами, дочерний процесс наследует свое местоположение. Это может вызвать проблему при размонтировании — система не позволит размонтировать раздел, на котором выполняется

демон. Чтобы этого избежать, оптимально переместить точку выполнения демона в место, которое гарантировано будет в системе — в root.

5. close(fd). Эта системная функция закрывает указанный файловый дескриптор. Демон не может работать со стандартными потоками ввода-вывода и отключен от терминала, поэтому все возможные дескрипторы необходимо закрыть. Ниже приведён код, который закрывает все возможные файловые дескрипторы.

```
for (int fd = 0; fd <= sysconf(_SC_OPEN_MAX); fd++) {
    close(fd);
}
```

6. dup2(fileno(log_file), fileno(stdout)). Важный шаг, который позволил упростить реализацию логирования. На этапе проектирования стояла следующая проблема: один и тот же код монитора должен использоваться как утилитой, так и демоном, но тогда как обеспечить вывод сообщений утилиты в терминал, а демона — в файл логирования? Аналогичная ситуация была с получением входных параметров. Было сделано такое решение: выводить логи всегда в стандартный поток вывода, но для демона переопределить его во время превращения программы в демона. Это и делает системный вызов dup2: устанавливает в соответствие одному файловому дескриптору другой. Таким образом, стандартным потоком вывода для демона становится файл логов. Аналогично стандартным потоком ошибок становится файл ошибок. Ниже приведен код, который устанавливает стандартный поток вывода для демона:

```
log_file = fopen(log_file_name, "a");
if (log_file == NULL || dup2(fileno(log_file), fileno(stdout)) == -1) {
    printf("can not open logs\n");
    return E_OPEN_LOGS;
}
```

7. fprintf(pid_file, "%d", getpid());. Для корректной работы демона как сервиса systemd, необходимо создать специальный pid-файл, который будет хранить pid демона. Эта операция также производится при

инициализации демона. После записи безопасным решением будет заблокировать этот файл на время работы демона во избежание случайных изменений. Ниже приведён код, который инициализирует pid-файл (путь к файлу берётся из аргументов запуска демона).

```
FILE* pid_file = fopen(pid_file_name, "w");
        if (pid_file == NULL) {
        exit(EXIT_FAILURE);
}
if (lockf(fileno(pid_file), F_TLOCK, 0) < 0) {
        exit(EXIT_FAILURE);
}
fprintf(pid_file, "%d", getpid());
fclose(pid_file);
```

Важным этапом в жизни демона является не только его рождение, но и смерть. Она также должна происходить без сбоев и побочных эффектов для операционной системы. Демон slmd подписан на два сигнала: SIGINT, который служит сигналом завершения, и SIGHUP, который служит сигналом перезагрузки файла конфигурации. При поступлении SIGINT сначала для всех зарегестрированных мониторов вызывается метод monitor_stop, а затем monitor_join и monitor_destroy. При поступлении сигнала SIGHUP происходит полное удаление всех мониторов и чтение файла конфигурации заново.

Для полноценной работы в качестве сервиса systemd, необходимо добавить конфигурационный файл демона в /usr/lib/systemd/system. Ниже приведено содержание файла slmd.service. Убедиться в его правильном распознавании системой systemd можно с помощью команды `systemctl list-unit-files | grep slmd`

```
[Unit]
Description=slm utility daemon version

[Service]
Type=forking
PIDFile=/var/run/slmd.pid
```

```
ExecStart=/usr/bin/slmd \
        --config-file /etc/config/slmd.conf \
        --log-file /var/log/slmd.log \
        --pid-file /var/run/slmd.pid
ExecReload=/bin/kill -HUP $MAINPID
KillSignal=SIGINT
ExecStop=/bin/kill -s SIGINT $MAINPID

[Install]
WantedBy=multi-user.target
```

После указанных операций демона пользователь может запустить демона с помощью команды `sudo systemctl start slmd`.

4.3 Общие принципы работы мониторов

Все созданный мониторы реализуют одинаковый интерфейс, объявленный в заголовочном файле monitor.h (рассмотрен выше). Каждый из них имеет свою структуру, которая используется для хранения необходимых данных (например, ссылки на поток).

Важной характеристикой монитора является его состояние, возможные состояние объявлены в заголовочном файле monitor.h. Мониторы используют это состояние, чтобы определить момент, в который необходимо завершить работу (MONITOR_STATE_DYING). Для того, чтобы не возникало коллизий доступа, использование переменной состояние везде обёрнуто в мьютекс.

Все созданные мониторы работают в отдельных потоках. Важно, что при создании потока он настраивается таким образом, чтобы игнорировать приходящие процессу системные сигналы и предоставить возможность главному потоку программы их обработать.

4.4 Работа inotify-монитора

inotify-монитор предназначен для отслеживания событий с файлами. Структура монитора представлена ниже.

```
struct inotify_monitor {
        int inotify_file_descriptor;
        char* file_path;
        char* mode;
        pthread_t thread;
        pthread_mutex_t state_mutex;
};
```

Алогоритм работы монитора следующий:

1. При инициализации монитора (метод inotify_monitor_from_args) выделятся память под монитор и его структура заполняется данными.
2. При запуске монитора создаётся новый поток. Этот процесс в цикле ожидает событий intotify с помощью вызова poll, периодически прерываясь на проверку состояния монитора. Если состояние сигнализирует о необходимости завершить работу, поток завершает выполнение.
3. При остановке монитора просто происходит переключение его состояние. Рабочий поток должен сам отследить его изменение и корректно завершиться (описано выше)

Схема алгоритма функции потока монитора представлена на чертеже ГУИР.400201.122 ПЗ.1.

4.5 Работа dbus-монитора

dbus-монитор предназначен для отслеживания состояния network manager и подключения дисков. Структура монитора представлена ниже.

```
struct dbus_monitor {
        int type;
        GMainLoop* subscription_loop;
        pthread_t thread;
        pthread_mutex_t state_mutex;
};
```

Алогоритм работы монитора следующий:

1. При инициализации монитора (метод dbus_monitor_from_args) выделятся память под монитор и его структура заполняется данными.
2. При запуске монитора создаётся новый поток. Этот процесс с помощью Glib-Dbus интерфейса регистрирует слушателей сигналов необходимых сигналов (для network manager достаточно одного, для мониторинга внешних накопителей необходимо два), а затем запускает цикл ожидания событий subscription_loop. При поступлении сигнала от dbus вызывается установленный обработчик, который и выводит нужную информацию в лог.

3. При остановке монитора просто происходит остановка subscription_loop. После остановки цикла поток корректно завершает работу.

Схема алгоритма функции потока монитора представлена на чертеже ГУИР.400201.122 ПЗ.2.

4.6 Работа udev-монитора

udev-монитор предназначен для отслеживания состояния питания и подключения bluetooth. Структура монитора представлена ниже.

```
struct z_udev_monitor {
        int type;

        pthread_t thread;
        pthread_mutex_t state_mutex;
};
```

Тип определяет, будет монитор работать с питанием или будет прослушивать изменения состояние bluetooth. Алогоритм работы монитора следующий:

1. При инициализации монитора (метод udev_monitor_from_args) выделятся память под монитор и его структура заполняется данными.
2. При запуске монитора создаётся новый поток. С помощью библиотеки libudev создаётся сначала экземпляр структуры udev, а затем по ней получается дескриптор прослушивания событий, пригодный для

использования в epoll. Также создаётся и настраивается libudev-монитор для отслеживания событий, зарегистрированных в udev по правилам (не kernel-события). Затем в цикле происходит опрос дескриптора событий на предмет поступления новых событий. Периодически происходит проверка состояния для возможности завершения работы.

3. При остановке монитора происходит переключение его состояние в MONITOR_STATE_DYING, а рабочий поток должен самостоятельно корректно завершиться.

Схема алгоритма функции потока монитора представлена на чертеже ГУИР.400201.122 ПЗ.3.

Весь исходный код программы доступен по ссылке https://github.com/ZoXaL/spovm4_slm

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Результат работы распространяется в виде пакета `slm-1.0zoxal1-Linux.deb`. Для его установки понадобятся права суперпользователя.

1. sudo dpkg -i slm-1.0zoxal1-Linux.deb

   Эта команда установит утилиты slm и slmd в систему.
2. Работа с slm
   1. Для просмотра помощи используется флаг --help
   2. Для работы с файлами используется флаг --file. Доступные флаги режима мониторинга (флаги можно объединять):
      - -o — мониторинг событий открытия файла.
      - -c — мониторинг событий закрытия файла.
      - -w — мониторинг событий изменения файла.
      - -d — мониторинг события удаления файла. Работа утилиты будет завершена после удаления отслеживаемого файла.

- -m — мониторинг события перемещения файла. Работа утилиты будет завершена после перемещения отслеживаемого файла.

3. Для отслеживания состояния network manager используется флаг --network.
4. Для отслеживания подключения внешних накопителей используется флаг --disks.
5. Для отслеживания состояния питания используется флаг --power.
6. Для отслеживания состояния bluetooth используется флаг — bluetooth.

3. Работа с slmd производится через systemctl (понадобятся права суперпользователя). Для того, чтобы указать, какие события демон должен отслеживать, необходимо в файле конфигурации /etc/config/slmd.config указать требуемые команды, по одной на каждую строчку. Команды аналогичны командам утилиты slm, за исключением того, что название утилиты писать не нужно. Напимер,

```
--disks
--file -owc /home/zoxal/my_file.txt
--power
```

Результаты работы демона можно отслеживать в файле /var/log/slmd.log.

1. sudo systemctl status slmd — просмотр состояния демона
2. sudo systemctl start slmd — запуск демона
3. sudo systemctl stop slmd — остановка демона
4. sudo systemctl reload slmd — перезагрузка файла конфигурации

ЗАКЛЮЧЕНИЕ

В результате выполнения данной курсовой работы был создан debian-пакет, содержащий две программы: утилиту slm и демон slmd, предназначенные для отслеживания системных событий Linux. При создании этих программ автором были получены навыки работы с новыми инструментами и библиотеками, такими как dbus, glib, udev, inotify, освоены принципы конструирования программы-демона в системе Linux и управления им с помощью менеджера служб systemd, основы создания debian-пакетов. Также освоена новая система сборки проектов на C/C++ CMake, навыки работы с которой наверняка пригодятся автору в будущем.

В архитектуру приложения заложены возможности дальнейшего развития и добавить поддержку нового монитора не составит большого труда.

Из возможных дальнейших усовершенствований программы можно предложить реализацию возможности запуска других приложений по настроенным событиям и передачу им типа произошедшего события.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. CMake tutorial [электронный ресурс]. Режим доступа: https://cmake.org/cmake-tutorial/
2. Linux daemon howto [электронный ресурс]. Режим доступа: http://www.netzmafia.de/skripten/unix/linux-daemon-howto.html
3. Getopt [электронный ресурс]. Режим доступа: https://www.gnu.org/software/libc/manual/html_node/Getopt.html
4. GVariatn documentation [электронный ресурс]. Режим доступа: https://developer.gnome.org/glib/stable/glib-GVariant.html
5. libudev and sysfs tutorial [электронный ресурс]. Режим доступа: http://www.signal11.us/oss/udev/
6. Python detection of usb stored device via DBus [электронный ресурс]. Режим доступа: https://linuxmeerkat.wordpress.com/2014/11/12/python-detection-of-usb-storage-device/
7. GDbusConnection documentation [электронный ресурс]. Режим доступа: https://developer.gnome.org/gio/stable/GDBusConnection.html
8. DBus tutorial [электронный ресурс]. Режим доступа: https://dbus.freedesktop.org/doc/dbus-tutorial.html

ПРИЛОЖЕНИЕ А

(обязательное)

Схема алгоритма inotify_monitoring_thread()

ПРИЛОЖЕНИЕ Б

(обязательное)

Схема алгоритма dbus_monitoring_thread()

ПРИЛОЖЕНИЕ В

(обязательное)

Схема алгоритма udev_monitoring_thread()