Министерство науки и образования Российской Федерации

ФГБОУ ВО

«Дагестанский государственный технический университет»

Методические указания к выполнению лабораторной работы №4 по дисциплине «Программирование в системах управления реального времени» для бакалавров направления 27.03.04- «Управление в технических системах»

Махачкала 2019

Разработка приложений реального времени для операционной системы Linux

Цель работы: Изучение принципов разработка приложений реального времени для операционной системы Linux..

1. Понятие «система реального времени»

Программная система является «системой реального времени», если успешность её работы зависит не только от логической правильности, но и от времени, за которое были получены результаты. Если такая система не может удовлетворить временным ограничениям, должен быть зафиксирован сбой в её работе. Стандарт POSIX 1003.1 определяет понятие «реальное время» как способность системы обеспечить требуемый уровень сервиса в определённый промежуток времени. Таким образом, предсказуемость времени реакции системы на непредсказуемое появление внешних событий является определяющей чертой систем реального времени.

Иногда понятие системы реального времени отождествляют с «быстрой системой», но это не всегда правильно, так как важно не время задержки реакции системы, а то, чтобы этого времени было достаточно для рассматриваемой задачи и оно было гарантированно. Во многих прикладных областях рассматривается своё понятие «реального времени». Рассмотрим пример из области цифровой обработки сигналов. Если при обработке аудио данных на анализ каждых T секунд звука требуется время, превышающее T, то подобный процесс обработки не является процессом реального времени. Если же требуется менее T секунд, то это уже процесс реального времени.

Различают системы «жёсткого» и «мягкого» реального времени. Система «жёсткого» реального времени гарантирует выполнение действий за определённый интервал времени. Обычно такие гарантии требуются для систем, для которых выход за пределы установленного срока реакции приводит к фатальному нарушению работоспособности системы.

Система «мягкого» реального времени, *как правило*, успевает выполнить действия за заданный промежуток времени. Для таких систем выход за пределы установленного срока приводит к уменьшению качества обработки, но не влияет на работоспособность системы в целом. В очень многих случаях программные системы ориентированы на «мягкое» реальное время.

Программные системы реального времени должны выполняться под управлением специальных операционных систем реального времени. Такие системы с помощью специальных средств обеспечивают поддержку функционирования в реальном времени. Набор подобных средств включает в себя:

- потоки управления;
- специальные методы планирования задач;
- сигналы реального времени;
- средства синхронизации;
- высокоточные таймеры;
- асинхронный ввод-вывод.

2. Реальное время и ОС Linux

Операционная система Linux обеспечивает поддержку перечисленныхвыше специальных средств для решения задач в реальном времени. Однако она, строго говоря, не является операционной системой реального времени.

При оценке систем реального времени используются две важнейшие характеристики:

• время ответа на прерывание – время между моментом выставления запроса на прерывание и моментом начала выполнения функции обработки прерывания;

• время ответа потока управления («latency») – время между моментом выставления запроса на прерывание и моментом начала выполнения потока, ответственного за реакцию на данное прерывание. Оно включает в себя, в частности, время ответа на прерывание, задержку планирования и время переключения контекста.

Для операционной системы Linux значение второй из указанных характеристик составляет 1-10 мс. То есть для тех случаев задач обработки и управления в реальном времени, когда требуемое время отклика составляет менее 1 миллисекунды, использование Linux становится проблематичным.

Вместе с тем многие задачи «мягкого» реального времени могут быть вполне успешно реализованы под управлением Linux. Приведённые оценки времён ответа справедливы для обычных пользовательских процессов. В то же время эти величины могут быть несколько уменьшены за счёт запуска приложения от имени суперпользователя системы. Это позволяет использовать ряд возможностей, таких как выбор схемы диспетчеризации, доступ к таймеру с высоким разрешением и принудительное размещение определённых данных в оперативной памяти без «свопинга», недоступных пользовательским процессам.

Запуск приложения от имени суперпользователя потенциально опасен с точки зрения безопасности системы, но во многих случаях такое решение является допустимым. Как правило, либо всё приложение реального времени, либо его ядро не предполагает какого-либо участия пользователя и может быть реализовано в виде автономной, самостоятельно функционирующей компоненты. Если же требуется контроль за состоянием системы и/или её управлением со стороны человека, то обеспечение соответствующих задач возлагается на обычное «пользовательское» приложение. Его связь с приложением реального времени организуется, используя стандартные средства межпроцессного взаимодействия, предоставляемые операционной системой.

Однако важно понимать, что Linux не предназначен для выполнения приложений «жёсткого» реального времени. Подобно всем операционным системам общего назначения, Linux пытается максимизировать показатель производительности в среднем, вместо рассмотрения наихудшего случая. В наихудшем случае производительность при обработке прерываний крайне невысокая. Большинство применённых методов повышения производительности приводит к уменьшению времени в «среднем случае», увеличивая время реакции в наихудшем случае. Вместе с тем для приложений «жёсткого» реального времени существуют специальные расширения к Linux, такие как RTLinux.

3. Многопоточная организация приложения

Практически все задачи реального времени могут быть представлены в виде набора подзадач, каждую из которых возможно решить или независимо от других подзадач, или с их минимальной кооперацией. При этом они выполняются конкурентно (в однопроцессорной системе) или параллельно в многопроцессорной системе. В многопоточной модели каждая такая подзадача существует как индивидуальный поток выполнения внутри одного и того же процесса. При этом процесс делится на две части. Одна часть содержит ресурсы, используемые через всю программу, такие как программный код и глобальные данные. Другая часть содержит информацию, относящуюся к состоянию выполнения, например, программный счетчик и стек. Эта часть называется *потоком* (thread).

В операционной системе Linux поддержка потоков обеспечена определенным набором типов языка программирования С и набором функций для выполнения операций над потоками. Поддержка потоков выполнения реализована в виде набора заголовочных файлов и библиотеки, подключаемой к программе на этапе ее компоновки. Важно понимать, что поток выполнения в операционной системе Linux представляется отдельным процессом, а не какой-либо частью уже существующего процесса.

При многопоточной организации работы приложения необходимо уметь управлять потоками (создание, завершение, блокирование), обеспечивать взаимоисключение при одновременном доступе к общим ресурсам и синхронизацию отдельных участков их работы с помощью примитивов взаимоисключения. При использовании библиотеки поддержки потоков предоставляется поддержка таких примитивов, как:

- мьютексы;
- семафоры;
- условные переменные.

Детально управление потоками и способы их взаимодействия с помощью примитивов взаимоисключения описываются в методических указаниях к лабораторной работе №8 «Разработка многопоточных приложений».

4. Ранжирование задач по приоритетам

Распределение времени центрального процессора между процессами (диспетчеризация) выполняется с учётом их приоритетов, числовой характеристики процесса, показывающей его «важность». Каждый процесс в Linux обладает статическим приоритетом в диапазоне 0..99. Значение статического приоритета процесса с заданным идентификатором *pid* можно узнать с помощью системного вызова *sched_getparam* со следующим прототипом:

*int sched_getparam(pid_t pid, struct sched_param *p);* – заполняет указанную структурную переменную типа *sched_param* значениями параметров диспетчеризации процесса с идентификатором *pid*. Значение приоритета может быть считано из поля *sched_priority* информационной структуры. В качестве результата возвращает признак успешности выполнения операции.

Прототип рассмотренного системного вызова, а также и других операций, связанных с ранжированием процессов по приоритетам, располагается в системном заголовочном файле *sched.h*

Обычные пользовательские процессы имеют нулевой статический приоритет. Их диспетчеризация осуществляется по стратегии разделения времени и основывается на динамическом приоритете. При такой стратегии время реакции процесса на наступление внешнего события может достигать 10 миллисекунд и для большинства задач реального времени не подходит.

Для процессов реального времени, требующих быстрый отклик на внешнее событие, используются ненулевые статические приоритеты. При этом процессы, готовые к выполнению, распределяются по спискам в соответствии с их значением статического приоритета. При определении процесса для выполнения диспетчер выбирает процесс из головы того непустого списка процессов, который имеет наибольший статический приоритет. Следует отметить, что статический приоритет может быть изменён только от имени суперпользователя. Изменение статического приоритета выполняется системным вызовом *sched_setparam* со следующим прототипом:

*int sched_setparam(pid_t pid, struct sched_param *p);* – устанавливает значение приоритета для указанного процесса *pid*. Необходимое значение приоритета должно быть предварительно занесено в поле *sched_priority* информационной структуры. В качестве результата возвращает признакуспешности выполнения операции.

Для процессов реального времени применяются специальные стратегии планирования реального времени:

• FIFO-планирование;

• RR-планирование.

Стратегия определяет правило помещения процесса в список процессов и его перемещение внутри списка. Она работает только при наличии конкуренции между процессами с одинаковыми приоритетами.

В стратегии «*FIFO-планирование*» процессор предоставляется процессам в порядке их поступления в список готовых. При этом процесс владеет процессором до своей блокировки или до появления в системе готового к выполнению процесса с более высоким приоритетом. При переходе в состояние готовности процесс помещается в конец списка. Если выполнение процесса прервано более приоритетным процессом, то он остаётся в голове списка. Данная стратегия реализует вытесняющую многозадачность в смысле наличия процессов с разными статическими приоритетами и совместную многозадачность с точки зрения процессов с одинаковым приоритетом. Такие процессы должны координировать свою деятельность, периодически блокируя своё выполнение, ожидая наступления внешнего события или выполняя операцию «уступка управления». В последнем случае процесс, не блокируя

себя, добровольно освобождает процессор другому готовому к выполнению процессу, имеющему тот же самый приоритет. Системный вызов для выполнения этой операции имеет следующий прототип:

int sched_yield(); – в качестве результата возвращает признак успешности выполнения операции.

При «*циклическом (RR) планировании*» время непрерывного владения процессором ограничено длительностью временного кванта (150 мкс). По истечении выделенного кванта процессор принудительно отнимается, а сам процесс перемещается в конец списка. Если выполнение процесса прервано более приоритетным процессом, то оставшуюся часть кванта он сможет отработать при первой же возможности. Данная стратегия реализует вытесняющую многозадачность.

При использовании любой из рассмотренных стратегий важно понимать, что готовый к выполнению единственный процесс с наибольшим статическим приоритетом полностью монополизирует процессор вычислительной системы на всё время до тех пор, пока каким-либо образом не перейдёт в состояние блокировки (ожидания наступления некоторого события).

Текущую стратегию планирования процесса с заданным идентификатором *pid* можно узнать с помощью системного вызова

sched_getscheduler со следующим прототипом:

int sched_getscheduler(pid_t pid); – в качестве результата возвращает

SCHED_FIFO, *SCHED_RR* или *SCHED_OTHER*. В случае ошибки возвращает -1.

Изменение стратегии планирования процесса выполняется системным вызовом *sched_setscheduler* со следующим прототипом:

*int sched_setscheduler(pid_t pid, int policy, struct sched_param *p);* – для процесса с идентификатором *pid* устанавливает в качестве текущей стратегию *policy* со значениями параметров из *p*. В случае ошибки возвращает -1.

Ниже приведён пример программы, устанавливающей у заданного процесса RR-стратегию планирования с требуемым значением приоритета.

```
//sched.cpp
#include <sched.h>
#include <unistd.h>
#include <iostream>
int main()
{
pid_t pid;
std::cout << "id процесса: ";
std::cin >> pid;
struct sched_param params;
int ret = sched_getparam(pid,&params);
if ( ret == -1 ) {
perror("sched");
return 1;
}
std::cout << "текущий приоритет: " << params.sched_priority << std::endl;
std::cout << "новый приоритет: ";
std::cin >> params.sched_priority;
ret = sched_setscheduler(pid,SCHED_RR,&params);
if ( ret == -1 ) {
perror("sched");
```

return 1;
}
}

5. Таймеры

Таймер является средством обеспечения задержек и измерения времени в вычислительной системе. Главной характеристикой таймера является его точность – минимальный гарантированно выдерживаемый интервал времени.

Очевидно, что в системах реального времени предпочтение следует отдавать средствам, обеспечивающим наиболее высокую точность.

Для измерения времени предпочтительно использовать системный вызов *gettimeofday* со следующим прототипом, определённым в файле *sys/time.h*:

*int gettimeofday(struct timeval *tv, struct timezone *tz);* – заносит в структурную переменную *tv* время, прошедшее с 1 января 1970 года.

Возвращает признак успешности выполнения операции. В качестве значения второго параметра следует всегда указывать 0, поскольку в настоящем он уже не используется и оставлен для совместимости.

Структурный тип *timeval* имеет следующий вид:

struct timeval {
time_t tv_sec; //секунды
suseconds_t tv_usec; //микросекунды
};

Одним из известных способов обеспечения задержек является использование программного таймера. Программные таймеры реализуются за счёт выполнения в цикле заданного количества одинаковых «пустых» операций. При фиксированной частоте работы процессора это позволяет точно определять прошедшее время. Главными минусами такого метода являются:

зависимость количества итераций цикла от типа и частоты процессора, невозможность выполнения других операций во время задержки. Последнее из перечисленного делает затруднительным успешное применение программных таймеров для приложений реального времени.

Проблему обеспечения задержек можно решить использованием аппаратных таймеров. Аппаратные таймеры функционируют независимо от центрального процессора и в момент срабатывания посылают прерывание.

Операционная система Linux для решения своих задач (например при планировании процессов) использует такой таймер для квантования времени.

Эти же кванты времени могут быть использованы и прикладными процессами для обеспечения задержек с помощью системного вызова *nanosleep*. Он имеет следующий синтаксис, определённый в файле *time.h*:

*int nanosleep(const struct timespec *req, struct timespec *rem);* –

приостанавливает выполнение программы *по крайней мере* на время в **req*.

Если же функция завершится раньше, то в **rem* будет занесено оставшееся время. Возвращает признак успешности выполнения операции.

Структурный тип *timespec* имеет следующий вид:

struct timespec {

time_t tv_sec; //секунды
long tv_nsec; //наносекунды(0..999 999 999)
};

Следует учитывать, что реальное разрешение используемого в ядре таймера составляет 1..10 мс в зависимости от используемой аппаратной платформы. Поэтому задержка может оказаться больше на величину разрешения таймера. По этой же причине при досрочном завершении системного вызова оставшееся время будет округлено в большую сторону.

Для процессов со стратегией планирования реального времени *nanosleep* поддерживает короткие высокоточные паузы. Задержки до 2 миллисекунд в этом случае будут обеспечиваться с микросекундной точностью. При этом используется программный таймер специального вида, независящий от параметров процессора и не занимающий полностью процессорное время системы.

Для обеспечения задержек, в том числе периодических, также можно использовать устройство «часы реального времени» («Real Time Clock»). Часы реального времени (IRQ 8) допустимо использовать для генерации сигналов с частотой от 2Гц до 8192Гц. Доступ к драйверу часов реального времени осуществляется через файл-устройство */dev/rtc*. Драйвер позволяет настроить работу устройства на необходимую частоту, вида $2n$. Процесс, использующий этот драйвер, получает данные из */dev/rtc* с соответствующей частотой. При этом считанное слово содержит в разрядах 8-31 число прерываний, сгенерированных с момента последнего считывания данных.

По умолчанию обычный пользователь (без прав суперпользователя) может использовать частоты только до 64Гц включительно. Для увеличения граничного значения частоты необходимо явное разрешение, используя от имени суперпользователя обращение к драйверу через файловую систему */proc* в виде:

echo частота | cat >/proc/sys/dev/rtc/max-user-freq

Однако если приложение реального времени выполняется от имени суперпользователя, то указанное ограничение несущественно.

Доступ к устройству осуществляется через стандартный системный вызов открытия файла *open()*. Полученный в результате файловый дескриптор используется во всех дальнейших операциях с открытым файлом. Управление часами реального времени выполняется посылкой запросов на устройство с помощью стандартного системного вызова *ioctl()*. Он имеет следующий синтаксис, определённый в файле *sys/ioctl.h*:

int ioctl(int d, int request, ...); – управление работой устройства, определяемого дескриптором *d* посылкой на него запроса *request*. Возвращает признак успешности выполнения операции (*-1* при возникновении ошибки) или результат выполнения запроса.

Каждое устройство определяет свой собственный набор допустимых запросов, причём некоторые из них требуют указания в *ioctl()* третьего параметра, уточняющего выполняемый запрос.

Для организации периодических задержек с помощью драйвера часов реального времени используются следующие запросы, определённые в *linux/rtc.h*:

• *RTC_IRQP_SET* – установка частоты прерываний от таймера, значение которой указывается в качестве третьего параметра;

• *RTC_PIE_ON* – активизация периодической генерации прерываний;

• *RTC_PIE_OFF* – остановка периодической генерации прерываний.

Чтение из устройства должно осуществляться по одному элементу типа *unsigned long*. При этом в старшие 24 разряда указанного элемента заносится количество прерываний, имевших место между двумя очередными операциями чтения. Если же прерываний не было, то попытка чтения приводит к блокированию процесса, выполнившего его, до возникновения очередного прерывания от устройства.

Ниже приводится иллюстрация использования часов реального времени.

```
//rtc_sample.cpp
#include <linux/rtc.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
int main()
{
int rtc = open("/dev/rtc",O_RDONLY);
if ( rtc == -1 )
return 1;
int frequency = 64; //частота генерации прерываний
int duration = frequency*10; //длительность работы программы
unsigned long counter = 0; //счётчик числа прерываний
//установка частоты прерываний
if ( ioctl(rtc,RTC_IRQP_SET,frequency) == -1 )
return 2;
//активизация таймера
if( ioctl(rtc,RTC_PIE_ON,0) == -1 )
return 3;
while ( counter < duration ) {
unsigned long rtcData;
//чтение данных из устройства
int ret = read(rtc,&rtcData,sizeof(rtcData));
if ( ret == -1 )
break;
counter += rtcData >> 8; //учёт общего числа прерываний
std::cout << counter << ' ' << (rtcData >> 8) << std::endl;
}
//остановка таймера
ioctl(rtc,RTC_PIE_OFF,0);
close(rtc);
}
```

6. Отображение в виртуальное адресное пространство

В современных операционных системах возможно отобразить содержимое файла или какой-либо его части в адресное пространство процесса, фактически в некоторую область памяти. После такой операции к содержимому файла можно получить доступ как к обычному массиву. Это эффективнее непосредственного использования системных вызовов *read* и *write*, поскольку загружаются только области памяти, которые реально используются в приложении. Это ещё более эффективно в случае интенсивного попеременного выполнения операций чтения и записи в пределах одной области файла.

Операция отображения части файла в адресное пространство процесса выполняется с помощью системного вызова *mmap()* со следующим прототипом, определённым в файле *sys/mman.h*:

*void *mmap(void *address, size_t length, int protect, int flags, int filedes, off_t offset)* – отображает участок со смещением *offset* байт длиной *length* байт открытого файла, заданного дескриптором *filedes*, в адресное пространство процесса по адресу *address*. В большинстве случае в качестве такого адреса указывается *NULL*, что заставляет саму операционную систему сформировать адрес отображения. Возвращает адрес, в который выполнено отображение файла или *-1* в случае ошибки. С помощью параметра *protect* определяются права доступа к области памяти. Возможные значения: *PROT_READ, PROT_WRITE, PROT_EXEC* и их комбинации.

С помощью параметра *flags* можно управлять механизмом отображения. При указании *MAP_PRIVATE* изменения сделанные в памяти не повлияют на содержимое самого файла. Процесс работает с копией содержимого, причём очевидно, что другие процессы не видят никаких изменений в файле. При указании *MAP_SHARED* изменения сделанные в памяти приводят к изменениям и в содержимом файла. При этом они сразу становятся доступными и другим процессам, работающим с этим же файлом. Важно отметить, что реальное изменение содержимого файла на диске может выполняться в любое время.

Отображение в память выполняется для полных страниц памяти. Это означает, что в качестве смещения и начального адреса должны указываться величины, кратные размеру страницы памяти.

Для принудительного реального изменения содержимого файла на диске может использоваться системный вызов *msync()* со следующим прототипом:

*int msync(void *address, size_t length, int flags)* – физическое изменение содержимого файла на диске, отображённого в память по адресу *address* размером *length* байт. С помощью параметра *flags* можно управлять выполнением этого действия. При указании *MS_SYNC* процесс не сможет продолжать выполнение до завершения изменений. При использовании

MS_ASYNC процесс только инициирует синхронизацию данных, но не ожидает её завершения. В любом случае операция синхронизации данных имеет смысл, только если отображение ранее было выполнено с флагом *MAP_SHARED*.

По завершении работы с участком памяти должна быть выполнена отмена отображения. Эта операция осуществляется системным вызовом *munmap()* со следующим прототипом:

*int munmap(void *addr, size_t length)* – отменяет ранее выполненное отображение в память по адресу *addr* размером *length* байт. Возвращает признак успешности выполнения операции.

Рассмотренный механизм отображения можно применять не только к регулярным файлам, но и к специальным, например к файл-устройствам. Например, существует файл-устройство с прямым доступом */dev/mem*, который представляет физическую память вычислительной системы. При этом элемент файла со смещением *offset* представляет ячейку оперативной памяти, расположенную по адресу *offset*. Используя механизм отображения в память применительно к данному файл-устройству, можно получить доступ из прикладной программы к любому адресу физической оперативной памяти.

Подобная возможность может оказаться очень полезной для организации взаимодействия с внешним устройством непосредственно из прикладной программы. При этом взаимодействие выполняется через память по адресам, ассоциированным с устройством. Такой подход во многих случаях позволяет избежать «посредника» между приложением и устройством – драйвера устройства. Однако, если требуется обрабатывать прерывания, приходящие с устройства, то драйвер становится необходимым, поскольку операционная система Linux не позволяет пользовательским процессам обрабатывать прерывания. Такая обработка возможна только в ядре операционной системы.

В приведённой ниже программе иллюстрируется непосредственный доступ к видеопамяти с адреса *0xa0000*:

```
//пример доступа к видеопамяти
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <iostream>
using namespace std;
int main()
{
int mem = open("/dev/mem",O_RDWR);
if ( mem == -1 ) {
cout << "нет доступа к устройству /dev/mem" << endl;
return 1;
}
unsigned char* video = (unsigned char*) mmap(0, 0x10000,
PROT_READ|PROT_WRITE, MAP_SHARED, mem, 0xa0000);
if ( video == (unsigned char*)(-1) ){
cout << "ошибка при доступе к видеопамяти" << endl;
return 2;
}
//работа с видеопамятью
//...
```

```
munmap(video,0x10000);
close(mem);
}
```

7. Блокировка страниц в оперативной памяти

Механизм виртуальной памяти предполагает использование раздела подкачки для временного хранения на диске данных пользовательских процессов. Тем самым достигается возможность для процессов «иметь памяти больше, чем есть на самом деле». При этом подкачка и выгрузка страниц памяти («page fault») выполняется ядром операционной системы незаметно для процессов и не требует их вмешательства. Однако для приложений реального времени этот момент очень важен. Подкачка страниц прозрачна для процесса только до тех пор, пока ему безразлично как долго выполняется простой доступ к памяти. Процессы реального времени не имеют возможности ждать. Также для них неприемлем широкий разброс по времени выполнения, возникающий вследствие случайности процесса подкачки страниц.

Для решения поставленной проблемы операционная система предоставляет механизм блокировки определённых страниц адресного пространства процесса в физической оперативной памяти. При этом гарантируется, что такие страницы никогда не будут выгружены в раздел подкачки и ситуация «page fault» для них не будет иметь места.

Блокировка страниц выполняется с помощью системного вызова *mlock()* со следующим прототипом, определённым в файле *sys/mman.h*:

*int mlock(const void *addr, size_t len)* – блокирует область виртуальных адресов начиная с *addr* длиной *len* байт процесса. Возвращает признак успешности выполнения запроса.

Страницы остаются заблокированными до тех пор, пока владеющий ими процесс явно не разблокирует их или не завершится. Разблокирование страниц выполняется с помощью системного вызова *munlock()* со следующим прототипом:

*int munlock(const void *addr, size_t len)* – разблокирует область виртуальных адресов начиная с *addr* длиной *len* байт процесса. Возвращает признак успешности выполнения запроса.

Во многих случаях представляется полезным заблокировать не только какие-либо области с данными, но все страницы виртуального адресного пространства процесса. Это можно выполнить системным вызовом *mlockall*:

int mlockall(int flags) – блокирует все области адресного пространства процесса. Параметр *flags* позволяет управлять блокированием страниц.

Установка разряда *MCL_CURRENT* заставляет заблокировать все текущие страницы адресного пространства. Указание же разряда *MCL_FUTURE* заставляет выполнять автоматическую блокировку всех страниц, которые будут выделяться процессу в будущем. Возвращает признак успешности выполнения запроса.

При выполнении этого системного вызова осуществляется блокирование страниц с программным кодом, данными и сегментом стека процесса, а также разделяемые библиотеки, разделяемая память и файлы, отображённые в память.

Разблокирование всех страниц процесса и снятие режима автоблокировки новых страниц выполняется системным вызовом *munlockall()* без параметров.

Как правило процесс, который блокирует свои страницы для повышения скорости выполнения, также использует и стратегию планирования реального времени. Кроме того важно понимать, что чем больше страниц памяти заблокировано, тем меньше остаётся «свободных» страниц в оперативной памяти. Это приводит к более частому возникновению запросов на подкачку страниц со стороны других процессов и может даже привести к ситуации невозможности запуска новых программ вследствие недостаточного свободного объёма памяти.

Из-за потенциального воздействия на другие процессы блокировку страниц позволено выполнять только процессам с привилегиями суперпользователя. Кроме этого система может устанавливать ограничения по объёму памяти, которую процесс может заблокировать.

Вследствие используемого в Linux механизма «копирование по записи» возможна ситуация, при которой обращение к заблокированным страницам вызовет в худшем случае операции ввода-вывода, а следовательно и непредвиденное замедление выполнения задачи. Поэтому кроме собственно блокировки страниц необходимо выполнять и предварительное обращение к ним.

Ниже приведён пример программы, использующей механизм блокирования страниц адресного пространства. Вызов функции *stackReserving* до выполнения блокировки позволяет расширить сегмент стека процесса до достаточных размеров, чтобы избежать в дальнейшем выделения новых страниц памяти возникающих вследствие необходимости расширения стека.

```
#include <sys/mman.h>
#include <algorithm>
const int StackDepth = 100000;
void stackReserving();
int main()
{
stackReserving();
mlockall(MCL_CURRENT);
//...
//Обработка
//...
munlockall();
}
void stackReserving()
{
int memory[StackDepth];
std::fill(memory,memory+StackDepth,0);
}
```

8. Ожидание готовности к вводу или выводу

В ряде случаев программе необходимо принимать и обрабатывать данные из нескольких источников сразу по мере их поступления. Например, к некоторой рабо-

чей станции ряд устройств подключен через асинхронный последовательный интерфейс и требуется быстрая реакция на поступление данных с этих устройств. Другой пример – процесс, который выступает в качестве сервера для совокупности других процессов, сообщение с которыми осуществляется через каналы или сокеты.

В подобных ситуациях невозможно использовать обычную операцию чтения данных *read*, поскольку этот системный вызов во-первых читает из одного устройства, а во-вторых блокирует выполнение процесса до появления данных в этом устройстве. При этом другие источники поступления данных не рассматриваются. Как один из вариантов решения указанной проблемы – использовать неблокирующее чтение и последовательно опрашивать различные устройства. Однако это решение крайне неэффективно из-за значительной загрузки процессора «пустой» работой.

Лучшее решение – использовать функцию *select*. Она блокирует выполнение процесса до появления готовности ввода или вывода на определённом наборе файловых дескрипторов или по истечении заданного временного интервала. Данная функция имеет следующий прототип, определённый в файле *sys/select.h*:

*int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)* – блокирует вызвавший процесс, ожидая перехода одного или нескольких файловых дескрипторов из наборов **readfds*, **writefds*, **exceptfds* в состояние «готовности», на время, не превышающее значение **timeout*. Дескриптор файла считается «готовым», если возможно выполнить соответствующую операцию ввода-вывода без блокировки процесса. В качестве параметра *nfds* указывается увеличенный на 1 максимальный номер дескриптора из всех трёх наборов. Возвращает общее число готовых дескрипторов, либо -1 при возникновении ошибки. В наборах после

завершения функции остаются лишь «готовые» дескрипторы. Набор **readfds* содержит дескрипторы для выполнения операции чтения, **writefds* – операции записи, **exceptfds* – для ожидания особых ситуаций.

Любой из наборов может отсутствовать, если в качестве значения соответствующего указателя передать *NULL*. Допустимо не задавать ограничения по времени выполнения ожидания путём указания в качестве последнего параметра *NULL*.

Набор файловых дескрипторов представляется специальным типом *fd_set*. При этом для работы с переменными этого типа определены следующие макроопределения:

*void FD_ZERO(fd_set *set);* – очищает набор *set*.
*void FD_SET(int fd, fd_set *set);* – добавляет дескриптор *fd* в набор *set*.
*void FD_CLR(int fd, fd_set *set);* – удаляет дескриптор *fd* из набора *set*.
*int FD_ISSET(int fd, fd_set *set);* – проверяет наличие дескриптора *fd* в наборе *set*.

В случае использования TCP-сокетов, для серверного сокета готовность по чтению означает поступление запроса на соединение. Для клиентского сокета готовность по записи означает установленное соединение с серверным сокетом.

Ниже приводится пример, иллюстрирующий использование функции *select*. Процесс ожидает поступления данных со стандартного устройства ввода в течение 5 секунд.

```
#include <stdio.h>
#include <unistd.h>
#include <iostream>
int main()
{
fd_set rfds;
struct timeval tv;
int retval;
//Ждать готовности stdin (fd 0)
FD_ZERO(&rfds);
FD_SET(0, &rfds);
//Ожидать в течение 5 секунд
tv.tv_sec = 5;
tv.tv_usec = 0;
retval = select(1, &rfds, NULL, NULL, &tv);
if (retval == -1)
perror("select()");
else if (retval)
std::cout << FD_ISSET(0, &rfds) << " Данные готовы" << std::endl;
else
std::cout << "Нет данных" << std::endl;
return 0;
}
```

9. Сигналы реального времени

Кроме набора стандартных сигналов процессы могут использовать дополнительный набор *сигналов реального времени*. Эти сигналы пронумерованы последовательно от *SIGRTMIN* до *SIGRTMAX*. Поскольку точные значения этих пределов могут варьироваться, то в программах всегда следует ссылаться на сигналы реального времени в виде *SIGRTMIN+n* и *SIGRTMAX-n*, а не использовать какое-либо конкретное числовое значение. В отличие от стандартных сигналов, сигналы реального времени не имеют предопределённого значения. Любой из них можно использовать для

своих целей. Однако, в многопоточных приложениях первые три из них задействованы для нужд средств поддержки потоков. Действием по умолчанию для сигналов реального времени является завершение процесса.

Сигналы реального времени имеют по сравнению со стандартными сигналами ряд различий:

• в очереди на обработку может находиться несколько сигналов одного вида (для стандартных – не более одного);

• при посылке сигнала можно дополнительно передать данные;

• доставка осуществляется в гарантированном порядке. Несколько сигналов одного вида доставляются процессу в том же порядке, в котором они были посланы. Если процессу посылаются различные сигналы, то первым будет доставлен сигнал с меньшим номером.

Определить обработчик сигнала или узнать его текущий способ обработки можно системным вызовом *sigaction*, имеющим следующий прототип, определённый в *signal.h*:

*int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);* – при ненулевом значении *act* системный вызов определяет способ обработки **act* сигнала *signum*, сохраняя информацию о предыдущем способе в структуре **oldact* при ненулевом значении *oldact*. Возвращает признак успешности выполнения операции. Структурный тип *sigaction* имеет следующий вид:

```
struct sigaction {
void (*sa_handler)(int);//обработчик без передачи данных
void (*sa_sigaction)(int, siginfo_t *, void *);//обработчик с данными
sigset_t sa_mask;//маска блокируемых при обработке сигналов
int sa_flags;//управление поведением процесса обработки сигнала
}
```

При необходимости передачи вместе с сигналом данных в поле *sa_flags* устанавливается разряд *SA_SIGINFO* и в поле *sa_sigaction* заносится адрес функции обработки сигнала. Такая функция принимает в качестве своих параметров номер сигнала, описание события ассоциированного с сигналом и некий указатель. По умолчанию определяемый обработчик действует многократно, однако указанием флага *SA_RESETHAND* разрешается однократная обработка.

Причину посылки сигнала можно установить по значению поля *si_code* структуры типа *siginfo_t*. При этом если источником сигнала стал процесс, то его идентификатор можно узнать по значению поля *si_pid* этой же структуры.

Если при посылке сигнала процессу были переданы данные, то *si_code* установлен в значение *SI_QUEUE*, а сами данные могут быть прочитаны из поля *si_value*.

Посылка сигнала процессом выполняется системным вызовом *sigqueue*, имеющим следующий прототип:

int sigqueue(pid_t pid, int sig, const union sigval value); – посылает процессу с номером *pid* сигнал *sig* совместно с данными, представленными значением *value*. Возвращает признак успешности выполнения операции.

Данные, передаваемые вместе с сигналом, имеют тип *union sigval* следующего вида:

```
union sigval {
int sival_int;
void *sival_ptr;
};
```

Таким образом вместе с сигналом процессу можно передать либо целочисленное значение, либо адрес некоторой области памяти. В последнем случае необходимо учитывать, что у каждого процесса своё собственное адресное пространство. Ниже приводится пример, иллюстрирующий приём-передачу сигналов реального времени. Процесс, принимающий сигнал, определяет обработчик и в цикле ожидает его многократного поступления. Процесс, посылающий сигнал, запрашивает дескриптор процесса-приёмника и циклически посылает ему соответствующий сигнал с данными, запрашиваемыми у пользователя.

```
//receiver.cpp
#include <signal.h>
#include <iostream>
//прототип обработчика
void handler(int sig, siginfo_t* info, void* data);
int main()
{
struct sigaction action;
action.sa_sigaction = handler;
sigemptyset(&action.sa_mask); //не блокировать сигналы при обработке
action.sa_flags = SA_SIGINFO; //приём данных вместе с сигналом
//задание способа обработки сигнала SIGRTMIN+3
sigaction(SIGRTMIN+3,&action,0);
while ( 1 )
pause();
}
void handler(int sig, siginfo_t* info, void* data)
{
std::cout << "сигнал " << sig;
std::cout << " доставлен от процесса " << info->si_pid;
std::cout << " с данными " << info->si_value.sival_int << std::endl;
}
//sender.cpp
#include <signal.h>
#include <iostream>
#include <unistd.h>
int main()
{
pid_t receiver;
std::cout << "задайте дескриптор процесса получателя:";
std::cin >> receiver;
while ( 1 ) {
sigval data;
std::cout << "задайте данные для посылки:";
std::cin >> data.sival_int;
if ( std::cin.eof() )
break;
sigqueue(receiver,SIGRTMIN+3,data); //посылка сигнала с данными
}
}
```

10. Получение системной информации

Процессам предоставляются средства для получения различной системной информации о параметрах операционной системы, о системных ограничениях и

ограничениях, накладываемых на процессы с возможностью изменения некоторых из них.

Для получения системной информации используется функция *sysconf,* прототип которой определён в *unistd.h* в следующем виде:

long sysconf(int name) – возвращает значение системного параметра, *name*.

С помощью этой функции можно получить значения системных констант, которые не меняются за время жизни процесса. Полный список допустимых значений параметра *name* приведён в странице руководства помощи по данной функции. Наиболее интересными из них являются следующие:

_SC_CLK_TCK – число тактов таймера в секунду, используемого ядром операционной системы;

_SC_PAGESIZE – размер страницы памяти в байтах;

_SC_CHILD_MAX – максимальное число существующих одновременно процессов, выполняемых от имени одного пользователя;

_SC_OPEN_MAX – максимальное число файлов, которые процесс может иметь открытыми;

_SC_PHYS_PAGES – число физических страниц в оперативной памяти;

_SC_NPROCESSORS_CONF – число сконфигурированных в системе процессоров.

Для получения информации об ограничениях в использовании ресурсов, накладываемых на процесс, можно использовать функцию *getrlimit* с прототипом, определённым в *sys/resource.h int getrlimit (int resource, struct rlimit *rlp)* – заполняет указанную структурную переменную **rlp* типа *rlimit* информацией о текущем и максимальном ограничениях использования ресурса *resource*. В качестве результата возвращает признак успешности выполнения операции.

Структурный тип *rlimit* имеет следующий вид:

```
struct rlimit {
rlim_t rlim_cur; //текущее («мягкое») ограничение
rlim_t rlim_max; // «жёсткое» ограничение
};
```

Значение поля *RLIM_INFINITY* означает отсутствие ограничения по ресурсу. Полный список допустимых ресурсов приведён в странице руководства помощи по данной функции. Наиболее интересными из них являются следующие:

RLIMIT_CPU – общее время использования процессора в секундах. При превышении «мягкого» ограничения процессу посылается сигнал *SIGXCPU*, а при превышении «жёсткого» ограничения – *SIGKILL*;

RLIMIT_MEMLOCK – максимальное число байт виртуальной памяти, которое может быть заблокировано в оперативной памяти;

RLIMIT_NPROC – максимальное число процессов, которые могут быть созданы от имени пользователя, являющегося владельцем вызвавшего процесса.

Процесс может установить свои ограничения на использование ресурсов.

Однако обычные процессы могут установить значение «мягкого» ограничения, не превышающее значение «жёсткого» ограничения, а также понизить значение последнего. Привилегированный процесс может произвольно менять любое из огра-

ничений. Для задания ограничений используется системный вызов *setrlimit* со следующим прототипом:

*int setrlimit (int resource, struct rlimit *rlp)* – использует информацию из структурной переменной **rlp* типа *rlimit* для изменения ограничения в использовании ресурса *resource*. В качестве результата возвращает признак успешности выполнения операции.

11. Контрольные вопросы и задания

1. Пусть в системе единственный процесс *A* имеет наибольшее значение статического приоритета и владеет процессором, а несколько процессов $B_1, \ldots B_k$ с меньшими значениями приоритета находятся в состоянии готовности. Кому выделит процессор диспетчер в результате выполнения операции «уступка управления» процессом *A*?
2. Имеется *n* источников информации. Что предпочтительнее: иметь *n* потоков выполнения, каждый из которых отвечает за свой единственный источник информации, или же один поток, ожидающий на нескольких источниках. Дайте обоснование ответа.
3. Обоснуйте или опровергните истинность следующего утверждения. «В некоторых случаях чтение процессом значения переменной приводит к выполнению операции записи (чтения) на жёстком диске».
4. Предложите способы формирования событий с частотой, не являющейся степенью двойки. Сопоставьте их между собой.
5. Приведите примеры задач, для которых операционную систему Linux можно применять и задач, для которых этого делать нельзя.
6. Предложите определение класса для манипулирования сигналами реального времени.
7. Предложите определение класса, обеспечивающего процессу доступ к требуемой области физических адресов оперативной памяти.

12. Список рекомендуемой литературы

1. Карпов В.Е., Коньков К.А. Основы операционных систем – М.: ИНТУИТ.ру, 2004. – 632 с.

2. Д. Бэкон, Т. Харрис. Операционные системы – СПб.: Питер; Киев: Издательская группа BHV, 2004. – 800 с.: ил.

3. Дейтел Г. Введение в операционные системы М.: Мир, 1987.

4. Теренс Чан. Системное программирование на C++ для Unix: Пер. С англ. – К.: Издательская группа BHV, 1997.