Лабораторная работа №5

Понятие о потоке (thread) в UNIX. Идентификатор потока. Функция pthread_self()

Цель: *Изучить работу потоков в операционной системе UNIX*

1. Теоретический блок

Во многих современных операционных системах существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора потоков (нитей исполнения). Потоки разделяют его программный код, глобальные переменные и системные ресурсы, но каждый поток имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у потоков являются общими, они могут использовать их как элементы разделяемой памяти.

В различных версиях операционной системы UNIX существуют различные интерфейсы, обеспечивающие работу с потоками. Мы кратко ознакомимся с некоторыми функциями, позволяющими разделить процесс на потоки и управлять их поведением, в соответствии со стандартом POSIX. Потоки, удовлетворяющие стандарту POSIX, принято называть POSIX thread'ами или, кратко, pthread'ами.

К сожалению, операционная система Linux не полностью поддерживает потоки на уровне ядра системы. При создании нового thread'а запускается новый традиционный процесс, разделяющий с родительским традиционным процессом его ресурсы, программный код и данные, расположенные вне стека, т. е. фактически действительно создается новый thread, но ядро не умеет определять, что эти thread'ы являются составными частями одного целого. Это «знает» только специальный процесс-координатор, работающий на пользовательском уровне и стартующий при первом вызове функций, обеспечивающих POSIX интерфейс для потоков. Поэтому мы сможем наблюдать не все преимущества использования потоков (в частности, ускорить решение задачи на однопроцессорной машине с

их помощью вряд ли получится), но даже в этом случае thread'ы можно задействовать как очень удобный способ для создания процессов с общими ресурсами, программным кодом и разделяемой памятью.

Каждый поток, как и процесс, имеет в системе уникальный номер — идентификатор thread'a. Поскольку традиционный процесс в концепции потоков трактуется как процесс, содержащий единственный поток, мы можем узнать идентификатор этого потока и для любого обычного процесса. Для этого используется функция *pthread_self()*. Поток, создаваемый при рождении нового процесса, принято называть *начальным* или *главным потоком* этого процесса.

Прототип функции *pthread_self()*

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Описание функции

Функция *pthread_self* возвращает идентификатор текущего потока. Тип данных *pthread_t* является синонимом для одного из целочисленных типов языка *C*.

Создание и завершение thread'a. Функции pthread_create(), pthread_exit(), pthread_join()

Потоки, как и традиционные процессы, могут порождать потоки-потомки, правда, только внутри своего процесса. Каждый будущий thread внутри программы должен представлять собой функцию с прототипом

*void *thread(void *arg);*

Параметр *arg* передается этой функции при создании thread'a и может, до некоторой степени, рассматриваться как аналог параметров функции main(). Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения

процесса-ребенка. Для создания нового потока применяется функция *pthread_create()*.

Прототип функции

#include <pthread.h>

*int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);*

Описание функции

Функция *pthread_create* служит для создания нового потока (thread'a) внутри текущего процесса. Настоящее описание не является полным описанием функции, а служит только целям данного курса. Для изучения полного описания обращайтесь к UNIX Manual.

Новый thread будет выполнять функцию *start_routine* с прототипом

*void *start_routine(void *)*

передавая ей в качестве аргумента параметр *arg*. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры. Значение, возвращаемое функцией *start_routine*, не должно указывать на динамический объект данного thread'a.

Параметр *attr* служит для задания различных атрибутов создаваемого thread'a. Их описание выходит за рамки нашего курса, и мы всегда будем считать их заданными по умолчанию, подставляя в качестве аргумента значение *NULL*.

Возвращаемые значения

При удачном завершении функция возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается положительное значение (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле *<ermo.h>*. Значение системной переменной *errno* при этом не устанавливается.

Созданный thread может завершить свою деятельность тремя способами:

• с помощью выполнения функции *pthread_exit()*. Функция никогда не возвращается в вызвавшую ее нить исполнения. Объект, на который указывает параметр этой функции, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread. Этот параметр, следовательно, должен указывать на объект, не являющийся локальным для завершившегося thread'a, например, на статическую переменную;

• с помощью возврата из функции, ассоциированной с нитью исполнения. Объект, на который указывает адрес, возвращаемый функцией, как и в предыдущем случае, может быть изучен в другой нити исполнения, например, в породившей завершившийся thread, и должен указывать на объект, не являющийся локальным для завершившегося thread'a;

• если в процессе выполняется возврат из функции *main()* или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции *exit()*, это приводит к завершению всех thread'ов процесса.

Функция для завершения потока

#include <pthread.h>

*void pthread_exit(void *status);*

Описание функции

Функция *pthread_exit* служит для завершения потока (thread) текущего процесса.

Функция никогда не возвращается в вызвавший ее thread. Объект, на который указывает параметр *status*, может быть впоследствии изучен в другой нити исполнения, например в нити, породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'a.

Одним из вариантов получения адреса, возвращаемого завершившимся thread'ом, с одновременным ожиданием его завершения является использование функции *pthread_join()*. Поток, вызвавший эту функцию, переходит в состояние ожидание до завершения заданного thread'a. Функция позволяет также получить указатель, который вернул завершившийся thread в операционную систему.

Прототип функции *pthread_join()*

#include <pthread.h>

*int pthread_join (pthread_t thread, void **status_addr);*

Описание функции

Функция *pthread_join* блокирует работу вызвавшего ее потока до завершения thread'a с идентификатором *thread*. После разблокирования в указатель, расположенный по адресу *status_addr*, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции *pthread_exit()*. Если нас не интересует, что вернул нам поток, в качестве этого параметра можно использовать значение *NULL*.

Возвращаемые значения

Функция возвращает значение 0 при успешном завершении. В случае ошибки возвращается положительное значение (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле *<errno.h>*. Значение системной переменной *errno* при этом не устанавливается.

Для иллюстрации вышесказанного давайте рассмотрим программу, в которой работают два потока:

/ Иллюстрация работы двух потоков. Каждый поток просто увеличивает на 1 разделяемую переменную a. */*

#include <pthread.h>

#include <stdio.h>

int a = 0 ;

/ Переменная a является глобальной статической для всей программы, поэтому она будет разделяться обоими потоками.*/*

/ Ниже следует текст функции, которая будет ассоциирована со 2-м thread'ом */*

*void *mythread(void *dummy)*

/ Параметр dummy в нашей функции не используется и присутствует только для совместимости типов данных. По той же причине функция возвращает значение void *, хотя это никак не используется в программе.*/*

{

pthread_t mythid; / Для идентификатора нити исполнения */*

/ Заметим, что переменная mythid является динамической локальной переменной функции mythread(), т. е. помещается в стеке и, следовательно, не разделяется нитями исполнения. */*

/ Запрашиваем идентификатор thread'a */*

mythid = pthread_self(); a = a+1;

printf("Thread %d. Calculation result = %d\n", mythid, a);

return NULL;

}

/ Функция main() - она же ассоциированная функция главного thread'a */*

int main() {

pthread_t thid, mythid; int result;

/ Пытаемся создать новую нить исполнения, ассоциированную с функцией mythread(). Передаем ей в качестве параметра значение NULL. В случае удачи в переменную thid занесется идентификатор нового thread'a. Если возникнет ошибка, то прекратим работу. */*

*result = pthread_create(&thid, (pthread_attr_t *)NULL, mythread, NULL);*

if(result 1= 0){

printf ("Error on thread create, return value = %d\n", result); exit(-1);

}

printf("Thread created, thid = %d\n", thid); / Запрашиваем идентификатор главного thread'a */*

mythid = pthread_self () ; a = a+1;

printf("Thread %d, Calculation result = %d\n",mythid, a);

/ *Ожидаем завершения порожденного thread'a, не интересуясь, какое значение он нам вернет. Если не выполнить вызов этой функции, то возможна ситуация, когда мы завершим функцию main() до того, как выполнится порожденный thread, что автоматически повлечет за собой его завершение, исказив результаты. */*

*pthread_join(thid, (void **)NULL); return 0;*

}

Для сборки исполняемого файла при работе редактора связей необходимо явно подключить библиотеку функций для работы с pthread'ами, которая не подключается автоматически. Это делается с помощью добавления к команде компиляции и редактирования связей параметра *-lpthread* — подключить библиотеку *pthread*.

Обратите внимание на отличие результатов этой программы от похожей программы, иллюстрировавшей создание нового процесса. Программа, создававшая новый процесс, печатала дважды одинаковые значения для переменной *a*, так как адресные пространства различных процессов независимы, и каждый процесс прибавлял 1 к своей собственной переменной *a*. Рассматриваемая программа печатает два разных значения, так как переменная *a* является разделяемой, и каждый thread прибавляет 1 к одной и той же переменной.

Необходимость синхронизации процессов и потоков, использующих общую память

Все рассмотренные на этом семинаре примеры являются не совсем корректными. В большинстве случаев они работают правильно, однако возможны ситуации, когда совместная деятельность этих процессов или нитей исполнения приводит к неверным и неожиданным результатам. Это связано с тем, что любые неатомарные операции, связанные с изменением содержимого разделяемой памяти, представляют собой критическую секцию процесса или нити исполнения.

При одновременном существовании двух процессов в операционной системе может возникнуть следующая последовательность выполнения операций во времени:

Процесс 1: array[0] += 1;

Процесс 2: array[1] += 1;

Процесс 1: array[2] += 1;

Процесс 1: printf("Program 1 was spawn %d times, program 2 - %d times, total - %d times\n", array[0], array[1], array[2]);

Тогда печать будет давать неправильные результаты. Естественно, что воспроизвести подобную последовательность действий практически нереально. Мы не сможем подобрать необходимое время старта процессов и степень загруженности вычислительной системы. Но мы можем смоделировать эту ситуацию, добавив в обе программы достаточно длительные пустые циклы перед оператором *array [2] + = 1*; Это проделано в следующих программах:

```
/* Иллюстрация некорректной работы с разделяемой памятью */
/* Мы организуем разделяемую память для массива из трех целых чисел.
Первый элемент массива является счетчиком числа запусков программы 1, т. е.
данной программы, второй элемент массива - счетчиком числа запусков
программы 2, третий элемент массива - счетчиком числа запусков обеих
программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
{
int *array; /* Указатель на разделяемую память */
int shmid; /* IPC-дескриптор для области разделяемой памяти */
```

```
int new =  1;   /* Флаг необходимости инициализации элементов массива  */
char pathname!]   =  "06-3a.c";   /* Имя файла,
```

*использующееся для генерации ключа. Файл с таким именем должен существовать в текущей директории */*

```
key_t key;   /*  IPC-ключ  */
long i;
/* Генерируем IPC-ключ из имени файла 06-3a.c в текущей директории и
```

*номера экземпляра области разделяемой памяти 0 */*

```
if((key = ftok(pathname,0))   < 0){ printf("Can\'t generate key\n");
exit(-1);
}
```

/ Пытаемся эксклюзивно создать разделяемую память для*

*сгенерированного ключа, т. е. если для этого ключа она уже существует, системный вызов вернет отрицательное значение. Размер памяти определяем как размер массива из трех целых переменных, права доступа 0666 - чтение и запись разрешены для всех */*

```
if((shmid = shmget(key,   3*sizeof(int),0666 | IPC_CREAT| IPC_EXCL) )   <
```

0){

/ В случае возникновения ошибки пытаемся определить: возникла ли она из-за того, что сегмент разделяемой памяти уже существует или по другой причине */*

```
if(errno   != EEXIST){
/* Если по другой причине - прекращаем работу */
printf("Can\'t create shared memory\n");
exit(-1) ;
}  else  {
```

/ Если из-за того,   что разделяемая память уже существует - пытаемся получить ее IPC-дескриптор и,   в случае удачи, сбрасываем флаг необходимости инициализации элементов массива */*

```
if((shmid = shmget(key,  3*sizeof(int),  0))  < 0){
```

```
 printf("Can\'t  find shared memory\n");
exit(-1);
}
new = 0;
}
}
/* Пытаемся отобразить разделяемую память в адресное пространство текущего процесса.  Обратите внимание на то,  что для правильного сравнения мы явно преобразовываем значение -1 к указателю на целое.*/
if((array =   (int  *)shmat(shmid,  NULL,   0))   == (int  *)(-1)){
printf("Can't attach shared memory\n");
exit(-1);
}
/* В зависимости от значения флага new либо инициализируем массив,  либо увеличиваем соответствующие счетчики */
if(new){
array[0]   =  1;
array[1]   = 0;
array[2]   =  1 ; }  else  {
array[0]   += 1;
for(i=0;   i<1000000000L;
/* Предельное значение для i может меняться в зависимости от производительности компьютера * array [2]   += 1;
}
/* Печатаем новые значения счетчиков,  удаляем разделяемую память из адресного пространства текущего процесса и завершаем работу */
printf("Program 1 was spawn %d times,
program 2  - %d times,  total - %d times\n", array[0],  array[1],  array[2]);
if(shmdt(array)   <  0){
printf("Can't detach shared memory\n");
```

```
exit(-1);
}
return 0;
}
/* Программа 2   (06-3b.c)  */
/* Мы организуем разделяемую память для массива  из трех целых чисел.
Первый элемент массива является счетчиком числа запусков программы 1, т.  е.
данной программы,   второй элемент массива - счетчиком числа запусков
программы 2,  третий элемент массива - счетчиком числа запусков обеих
программ */
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <errno.h>
int main()
(
int  *array;   /* Указатель на разделяемую память */
int shmid;   /*  IPC-дескриптор для области
разделяемой памяти  */
int new =  1;   /* Флаг необходимости инициализации
элементов массива  */
char pathname[]   =  "0б-3a.c";   /* Имя файла, использующееся для
генерации ключа.  Файл с таким именем должен существовать в текущей
директории */
key_t  key;   / *  IPC-ключ  */
long i;
/*  Генерируем IPC-ключ из  имени файла  0б-3a.c  в текущей директории и
номера экземпляра области разделяемой памяти 0  */
if((key =  ftok(pathname,0))   <  0){
```

```
printf("Can\'t generate key\n");
exit ( -1) ;
}
/* Пытаемся эксклюзивно создать разделяемую память для
сгенерированного ключа, т. е. если для этого ключа она уже существует,
системный вызов вернет отрицательное значение. Размер памяти определяем
как размер массива из трех целых переменных, права доступа 0666 - чтение и
запись разрешены для всех */
if((shmid = shmget(key, 3*sizeof(int),
0666| IPC_CREAT|IPC_EXCL) ) < 0){
/* В случае ошибки пытаемся определить, возникла ли она из-за того,
что сегмент разделяемой памяти уже существует или по другой причине */
if(errno != EEXIST){
/* Если по другой причине - прекращаем работу */
printf("Can\'t create shared memory\n");
exit(-1);
} else {
/* Если из-за того, что разделяемая память уже существует - пытаемся
получить ее IPC-дескриптор и, в случае удачи, сбрасываем флаг необходимости
инициализации элементов массива */
if((shmid = shmget(key, 3*sizeof(int) , 0)) < 0) {
printf("Can\'t find shared memory\n");
exit(-1);
}
new = 0;
}
}
/* Пытаемся отобразить разделяемую память в адресное пространство
текущего процесса. Обратите внимание на то, что для правильного сравнения
мы явно преобразовываем значение -1 к указателю на целое.*/
```

```
if((array =   (int  *)shmat(shmid,  NULL,   0))
== (int  *) (-1)) (
printf("Can't attach shared memory\n");
exit(-1);
}
/* В зависимости от значения флага new либо инициализируем массив,  либо
увеличиваем соответствующие счетчики */
if(new)(
array[0]   = 0;
array[1]   = 1;
array [2]   = 1;
}  else  {
array[1]   += 1;
for(i=0;   i<1000000000L;   i + +);
/*  Предельное  значение  для  i  может  меняться  в  зависимости  от
производительности компьютера */
array[2]   += 1;
}
/* Печатаем новые значения счетчиков, удаляем разделяемую память из
адресного пространства текущего процесса и завершаем работу */
printf("Program 1 was spawn %d times, program 2  - %d times,  total - %d
times\n", array[0],  array[1],  array[2]);
if(shmdt(array)   < 0){
printf("Can't detach shared memory\n");
exit(-1);
}
return 0;
}
```

Наберите программы, сохраните под именами 06-3a.c и 06-3b.c соответственно, откомпилируйте их и запустите любую из них один раз для создания

и инициализации разделяемой памяти. Затем запустите другую и, пока она находится в цикле, запустите (например, с другого виртуального терминала) снова первую программу. Вы получите неожиданный результат: количество запусков по отдельности не будет соответствовать количеству запусков вместе.

Как мы видим, для написания корректно работающих программ необходимо обеспечивать взаимоисключение при работе с разделяемой памятью и, может быть, взаимную очередность доступа к ней. Это можно сделать с помощью алгоритмов синхронизации, например, алгоритма Петерсона или алгоритма булочной.

2. Практические задания

Задание 1. Написание программы с использованием трех нитей исполнения

Напишите программу, имеющую 3 потока, увеличивающих значение разделяемой переменной на 1.

Задание 2. Синхронизация работы потоков

Модифицируйте программы из этого раздела для корректной работы с помощью алгоритма Петерсона.

Литература

1. В.Е. Карпов, К.А. Коньков Основы операционных систем. Курс лекций. Учебное пособие. Издание второе, дополненное и исправленное. М.: ООО «ИНТУИТ.ру» 2005