**Лабораторная работа 7-8**

**Синхронизация процессов/потоков. Передача данных между процессами/потоками**

Цель работы - практическое освоение механизмов синхронизации процессов и их посредством механизмов, предоставляемых ОС.

**Лабораторная работа 7** предполагает решение одной из классических задач синхронизации в конкретной операционной среде.

1. Реализация задачи «поставщик-потребитель». Требуется реализовать приложение- поставщик и приложение-потребитель или многопоточное приложение с потоком «поставщик» и потоком «потребитель».

2. Реализация задачи «читатели-писатели» (аналогично).

3. Реализация задачи «обедающие философы». Требуется реализовать многопоточное приложение с параметром «число философов».

4. Реализация задачи «спящий парикмахер» (аналогично).

**Лабораторная работа 8** подразумевает создание приложений (или одного многопоточного приложения), которые не только синхронизуются, но и обмениваются данными.

Работа имеет следующие варианты заданий:

1. Имеется поток (сервер), обрабатывающий запросы другого потока (клиента). Сервер в один момент времени может обрабатывать только один запрос. Клиент не должен посылать следующий запрос, не дождавшись ответа сервера. В качестве запроса посылается номер стоки файла, содержимое которого сервер возвращает в качестве ответа. Предполагаются следующие варианты задачи: синхронизация с помощью семафоров, передача запроса - ответа через общую память или через программные каналы.

2. Процесс писатель записывает содержимое некоторого файла. Процессы - читатели считывают данные, записанные процессом - писателем. Необходимо обеспечить взаимное исключение доступа к данным писателя и любого из читателей. Предполагаются следующие варианты задачи: синхронизация с помощью семафоров, обмен данными через общую память, программные каналы или очереди.

3. Обедающие философы (вариация). Создается пять процессов (по одному на философа). Процессы разделяют пять переменных (вилок). Каждый процесс находится только в двух состояниях - либо он "размышляет", либо "ест спагетти". Чтобы начать "есть", процесс должен взять "две вилки" (захватить две переменные). Закончив "еду", процесс освобождает захваченные переменные и начинает "размышлять" до тех пор, пока снова "проголодается". Предполагается синхронизация с помощью семафоров.

4. Исходный процесс порождает два процесса Р1 и Р2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс Р2 помещает в канал К1, затем они оттуда читаются процессом Р1, переписываются в канал К2, дополняются своими данными. Обработка данных основным процессом заключается в чтении информации из программного канала К2 и печати ее.

5. Исходный процесс создает два программных канала К1 и К2 и порождает новый процесс Р1, а тот, в свою очередь, еще один процесс Р2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс Р1 помещает в канал К1, а процесс Р2 в канал К2, откуда они процессом Р1 копируются в канал К1 и дополняются новой порцией данных. Обработка данных основным процессом заключается в чтении информации из программного канала К1 и печати ее.

6. Исходный процесс создает программный канал К1 и порождает новый процесса Р1, а тот, в свою очередь, порождает еще один процесс Р2. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Файл, читаемый процессом Р2, должен быть достаточно велик с тем, чтобы его чтение не завершилось ранее, чем закончится установленная задержка в n секунд. После срабатывания будильника процесс Р1 посылает сигнал процессу Р2, прерывая чтение файла. Обработка данных основным процессом заключается в чтении информации из программного канала и печати ее.

В ходе выполнения лабораторной работы должен:

1. Ознакомиться с заданием к лабораторной работе.

2. Выбрать набор примитивов синхронизации и реализующих их системных вызовов, обеспечивающих решение задачи.

3. Для указанного варианта составить программу на языке Си, реализующую требуемые действия.

4. Отладить и протестировать составленную программу, используя инструментарий ОС UNIX.

5. Защитить лабораторную работу, ответив на контрольные вопросы и написав отчет.

***Механизмы межпроцессного взаимодействия ОС UNIX***

Предполагается, что слушателям знакомы программные средства, связанные с созданием и управлением процессами в рамках ОС UNIX. Данная лабораторная работа предполагает комплексное их использование при решении задачи синхронизации процессов и их взаимодействия посредством различных механизмов, предоставляемым ОС.

Кратко перечислим состав системных вызовов, требуемых для выполнения лабораторных работ:

1. Создание, завершение процесса, получение информации о процессе, - fork(), exit(), getpid(), getppid();

2. Синхронизация процессов - signal(), kill(), sleep(), alarm(), wait(), pause();

3. Создание информационного канала и работа с ним - pipe(), read(), write().

Механизм IPC (Inter-Process Communication Facilities) включает:

- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам (семафоры - semaphores);

- средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу (очереди сообщений - message queries);

- средства, обеспечивающие возможность наличия общей для процессов памяти (сегменты разделяемой памяти - shared memory segments).

Наиболее общим понятием IPC является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип key\_t, состав которого зависит от реализации и определяется в файле <sys/types.h>. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту. Обе операции выполняются посредством операции get. Результатом операции get является его целочисленный идентификатор, который может использоваться в других функциях межпроцессного взаимодействия.

**Семафоры.**

Для работы с семафорами поддерживаются три системных вызова:

- semget() для создания и получения доступа к набору семафоров;

- semop() для манипулирования значениями семафоров (это тот системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров

- semctl() для выполнения разнообразных управляющих операций над набором семафоров

Прототипы перечисленных системных вызовов описаны в файлах

#include <sys/ipc.h>

#include <sys/sem.h>

Системный вызов semget() имеет следующий синтаксис:

semid = int semget(key\_t key, int count, int flag);

параметрами которого является ключ (key) набора семафоров и дополнительные флаги (flags), определенные в <sys/ipc.h>, число семафоров в наборе семафоров (count), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров semid. После вызова semget() индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе. Флаги системного вызова semget() приведены ниже в таблице.

|  |  |
| --- | --- |
| Таблица. Флаги системного вызова semget()  IPC\_CREAT | Semget создает новый семафор для данного ключа. Если флаг IPC\_CREAT не задан, а набор семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров. |
| IPC\_EXLC | Флаг IPC\_EXLC вместе с флагом IPC\_CREAT предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, Semget возвратит -1, а системная переменная errno будет содержать значение EEXIST. |

Младшие 9 бит флага задают права доступа к набору семафоров.

Системный вызов semctl() имеет формат

int semctl (int semid, int sem\_num, int command, union semun arg),

где semid - это идентификатор набора семафоров, sem\_numb - номер семафора в группе, command - код операции, а arg - указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции.

Структура msg имеет вид:

union semun

{

int val;

struct semid\_ds \*buf;

unsigned short \*array;

};

С помощью semctl() можно

- уничтожить набор семафоров или индивидуальный семафор в указанной группе (IPC\_RMID);

- вернуть значение отдельного семафора (GETVAL) или всех семафоров (GETALL);

- установить значение отдельного семафора (SETVAL) или всех семафоров (SETALL);

- вернуть число семафоров в наборе семафоров (GETPID).

Основным системным вызовом для манипулирования семафором является

int semop (int semid, struct sembuf \*op\_array, count),

где semid - это ранее полученный дескриптор группы семафоров, op\_array - массив структур sembuf, определенных в файле <sys/sem.h.h> и содержащих описания операций над семафорами группы, а count - размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива op\_array имеет следующую структуру (структура sembuf):

- номер семафора в указанном наборе семафоров

- операция над семафором;

- флаги.

Если указанные в массиве op\_array номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов последовательно меняет значение семафора (если это возможно) в соответствии со значением поля "операция". Возможны три случая:

1. Отрицательное значение sem\_op.

Если значение поля операции sem\_op отрицательно, и его абсолютное значение меньше или равно значению семафора semval, то ядро прибавляет это отрицательное значение к значению семафора.

Если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора.

Если же значение поля операции sem\_op по абсолютной величине больше семафора semval, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

2. Положительное значение sem\_op.

Если значение поля операции sem\_op положительно, то оно прибавляется к значению семафора semval, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии UNIX).

3. Нулевое значение sem\_op.

Если значение поля операции sem\_op равно нулю, то если значение семафора semval также равно нулю, выбирается следующий элемент массива op\_array.

Если же значение семафора semval отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания

При использовании флага IPC\_NOWAIT ядро ОС UNIX не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей бы к блокированию процесса при отсутствии флага IPC\_NOWAIT.

**Очереди сообщений.**

Для обеспечения возможности обмена сообщениями между процессами механизм очередей поддерживается следующими системными вызовами:

msgget() для образования новой очереди сообщений или получения дескриптора существующей очереди;

msgsnd() для постановки сообщения в указанную очередь сообщений;

msgrcv() для выборки сообщения из очереди сообщений;

msgctl() для выполнения ряда управляющих действий

Прототипы перечисленных системных вызовов описаны в файлах

#include <sys/ipc.h>

#include <sys/msg.h>

По системному вызову msgget() в ответ на ключ (key) и набор флагов (полностью аналогичны флагам в системном вызове semget()) ядро либо создает новую очередь сообщений и возвращает пользователю идентификатор созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий идентификатор очереди:

int msgqid = msgget (key\_t key, int flag).

Для помещения сообщения в очередь служит системный вызов msgsnd():

int msgsnd (int msgqid, void \*msg, size\_t size, int flag),

где msg - это указатель на структуру длиной size, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение.

Структура msg имеет вид:

struct msg {

long mtype; /\* тип сообщения \*/

char mtext[SOMEVALUE]; /\* текст сообщения (SOMEVALUE - любое \*/};

Параметр flag определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти (флаг IPC\_NOWAIT со значением, рассмотренным выше).

Условиями успешной постановки сообщения в очередь являются:

- наличие прав процесса по записи в данную очередь сообщений;

- не превышение длиной сообщения заданного системой верхнего предела;

- положительное значение указанного в сообщении типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди сообщений (флаг IPC\_NOWAIT при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений.

Для приема сообщения используется системный вызов msgrcv():

int msgrcv (int msgqid, void \*msg, size\_t size, long msg\_type, int flag);

Системный вызов msgctl()

int msgctl (int msgqid, int command, struct msqid\_ds \*msg\_stat)

используется

- для опроса состояния описателя очереди сообщений (command = IPC\_STAT) и помещения его в структуру msg\_stat (детали опускаем);

- изменения его состояния (command = IPC\_SET), например, изменения прав доступа к очереди;

- для уничтожения указанной очереди сообщений (command = IPC\_RMID).

**Работа с разделяемой памятью.**

Для работы с разделяемой памятью используются системные вызовы:

shmget() создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;

shmat() подключает сегмент с указанным описателем к виртуальной памяти обращающегося процесса;

shmdt() отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;

shmctl() служит для управления разнообразными параметрами, связанными с существующим сегментом.

Прототипы перечисленных системных вызовов описаны в файлах

#include <sys/ipc.h>

#include <sys/shm.h>

После того, как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи.

Системный вызов

int shmid = shmget (key\_t key, size\_t size, int flag)

на основании параметра size определяет желаемый размер сегмента в байтах. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Создание сегмента не означает немедленного выделения под него основной памяти и это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Флаги IPC\_CREAT и IPC\_EXCL аналогичны рассмотренным выше.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову shmat():

void \*virtaddr = shmat(int shmid, void \*daddr, int flags).

Параметр shmid - это ранее полученный идентификатор сегмента, а daddr - желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является фактический виртуальный адрес начала сегмента. Если значением daddr является NULL, ядро выбирает наиболее удобный виртуальный адрес начала сегмента. Флаги системного вызова shmat() приведены ниже в таблице.

|  |  |
| --- | --- |
| Таблица Флаги системного вызова shmat() SHM\_RDONLY | ядро подключает участок памяти только для чтения; |
| SHM\_RND | определяет, если возможно, способ обработки ненулевого значения daddr. |

Для отключения сегмента от виртуальной памяти используется системный вызов shmdt():

int shmdt(\*daddr);

где daddr - это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова shmat().

Системный вызов shmctl:

int shmctl (int shmid, int command, struct shmid\_ds \*shm\_stat);

по синтаксису и назначению системный вызов полностью аналогичен msgctl().