Семинары 11-12. Очереди сообщений

Сообщения как средства связи и средства синхронизации процессов

В материалах предыдущих семинаров были представлены такие средства организации взаимодействия процессов из состава средств System V IPC, как разделяемая память (семинары 7–8) и семафоры (семинары 9-10). Третьим и последним, наиболее семантически нагруженным средством, входящим в System V IPC, являются *очереди сообщений*. В лекции 6 говорилось о модели сообщений как о способе взаимодействия процессов через линии связи, в котором на передаваемую информацию накладывается определенная структура, так что процесс, принимающий данные, может чётко определить, где заканчивается одна порция информации и начинается другая. Такая модель позволяет задействовать одну и ту же линию связи для передачи данных в двух направлениях между несколькими процессами. Мы также рассматривали возможность использования сообщений с встроенными механизмами взаимоисключения и блокировки при чтении из пустого буфера и записи в переполненный буфер для организации синхронизации процессов. Рассмотренные на лекции очереди сообщений, использующие примитивы **send** и **receive** для приёма и передачи информации, принято называть *классическими очередями сообщений*.

На этом семинаре вы познакомитесь с реализацией очередей сообщений в System V IPC.

Отличия очередей сообщений System V IPC от классических очередей сообщений

Сообщения в System V IPC, в отличие от классических очередей, имеют дополнительный атрибут, называемый типом сообщения. Тип сообщения — это целое положительное число. Наличие у сообщения типа позволяет организовывать чтение сообщений из очереди (выборку сообщений) с помощью разных политик, в отличие от классических очередей сообщений, где реализуется только одна политика — FIFO.

Выборка сообщений из очереди (выполнение примитива **receive**) может осуществляться тремя способами:

1. В порядке FIFO, независимо от типа сообщения.
2. В порядке FIFO для сообщений конкретного типа.
3. Выборка с порогом. Первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения (порога), пришедшее ранее всех других сообщений с тем же типом.

Очереди сообщений в UNIX как составная часть System V IPC

Так как очереди сообщений входят в состав средств System V IPC, для них верно всё, что говорилось ранее об этих средствах в целом, и уже знакомо вам. Очереди сообщений, как и семафоры, и разделяемая память, являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действий для освобождения системных ресурсов по окончании взаимодействия. Пространством имён очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции *ftok()*. Для выполнения примитивов **send** и **receive**, введенных в лекции 6, соответствующим системным вызовам в качестве параметра передаются IPC-дескрипторы очередей сообщений, однозначно идентифицирующих их во всей вычислительной системе.

Очереди сообщений располагаются в адресном пространстве ядра операционной системы в виде однонаправленных списков и имеют ограничение по объему информации, хранящейся в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение.

Реализация примитивов **send** и **receive** обеспечивает скрытое от пользователя взаимоисключение во время помещения сообщения в очередь или его получения из очереди. Также она обеспечивает блокировку процесса при попытке выполнить примитив **receive** над пустой очередью или очередью, в которой отсутствуют сообщения запрошенного типа, или при попытке выполнить примитив **send** для очереди, в которой нет свободного места.

Очереди сообщений, как и другие средства System V IPC, позволяют организовать взаимодействие процессов, не находящихся одновременно в вычислительной системе.

Создание очереди сообщений или доступ к уже существующей. Системный вызов msgget()

Для создания очереди сообщений, ассоциированной с определённым ключом, или доступа по ключу к уже существующей очереди используется системный вызов *msgget()*, являющийся аналогом системных вызовов *shmget()* для разделяемой памяти и *semget()* для массива семафоров, который возвращает значение IPC-дескриптора для этой очереди. При этом существуют те же способы создания и доступа, что и для разделяемой памяти или семафоров. Подробное описание смотрите в справочнике man.

Реализация примитивов send и receive. Системные вызовы msgsnd() и msgrcv()

Для выполнения примитива **send** используется системный вызов *msgsnd()*, копирующий пользовательское сообщение в очередь сообщений, заданную IPC-дескриптором. При изучении описания этого вызова обратите особое внимание на следующие моменты:

```
Прототип системного вызова

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>



int msgsnd(int msqid, struct msgbuf *ptr, int length, int flag);
```

- Тип данных **struct msgbuf**, используемый во втором параметре, не является типом данных для пользовательских сообщений, а представляет собой лишь шаблон для создания таких типов. Пользователь сам должен создать структуру для своих сообщений, в которой первым полем должна быть переменная типа **long**, содержащая положительное значение типа сообщения.
- В качестве третьего параметра – длины сообщения – указывается не вся длина структуры данных, соответствующей сообщению, а только длина полезной информации, т. е. информации, располагающейся в структуре данных после типа сообщения. Это значение может быть и равным 0 в случае, когда вся полезная информация заключается в самом факте прихода сообщения (сообщение используется как сигнальное средство связи).
- В материалах семинаров вы, как правило, будете использовать нулевое значение флага системного вызова (четвёртый параметр), которое приводит к блокировке процесса при отсутствии достаточного свободного места в очереди сообщений.

Примитив **receive** реализуется системным вызовом *msgrcv()*. При изучении описания этого вызова нужно обратить особое внимание на следующие моменты:

```
Прототип системного вызова

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>



int msgrcv(int msqid, struct msgbuf *ptr, int length, long type, int flag);
```

- Тип данных **struct msgbuf**, как и для вызова *msgsnd()*, является лишь шаблоном для пользовательского типа данных.
- Политика выборки сообщения задаётся четвёртым параметром **type**. Нулевое значение параметра соответствует политике выборки FIFO независимо от типа сообщения, положительное значение параметра n соответствует второй политике выборки для типа n, отрицательное значение параметра соответствует политике выборки с порогом, где значение порога есть модуль параметра. Точное значение типа выбранного сообщения при первой или третьей политике выборки можно определить из соответствующего поля структуры (адрес, указанный во втором параметре), в которую системный вызов скопирует сообщение.
- Системный вызов возвращает длину только полезной части скопированной информации, т. е. информации, расположенной в структуре после поля типа сообщения.
- Выбранное сообщение удаляется из очереди сообщений.
- В качестве параметра **length** указывается максимальная длина полезной части информации, которая может быть размещена в структуре, адресованной параметром **ptr**.
- В материалах семинаров вы будете, как правило, пользоваться нулевым значением флагов (пятый параметр) для системного вызова, которое приводит к блокировке процесса в случае отсутствия в очереди сообщений с запрошенным типом и к ошибочной ситуации в случае, когда длина информативной части выбранного сообщения превышает длину, специфицированную в параметре **length**.

Максимально возможная длина сообщения и максимальный размер очереди сообщений в операционной системе Linux задаётся при генерации системы. Текущие значения максимальных длин можно определить с помощью команды:

```
$ ipcs -l
```

Удаление очереди сообщений из системы с помощью команды ipcrm или системного вызова msgctl()

После завершения процессов, использовавших очередь сообщений, она не удаляется из системы автоматически, а продолжает сохраняться в системе вместе со всеми невостребованными сообщениями до тех пор, пока не будет выполнена специальная команда или специальный системный вызов. Для удаления очереди сообщений можно воспользоваться уже знакомой вам командой *ipcrm*, которая в этом случае примет вид:

```
$ ipcrm msg <IPC идентификатор>
```

Для получения IPC-идентификатора очереди сообщений примените команду *ipcs*. Можно удалить очередь сообщений и с помощью системного вызова *msgctl()*. Этот вызов умеет выполнять и другие операции над очередью сообщений, но в рамках данного курса мы их рассматривать не будем. Если какой-либо процесс находился в состоянии **ожидание** при выполнении системного вызова *msgrcv()* или *msgsnd()* для удаляемой очереди, то он будет разблокирован, и системный вызов констатирует наличие ошибочной ситуации.

Для иллюстрации сказанного рассмотрим простые программы **ex1a.c** и **ex1b.c**.

Первая из этих программ посылает пять текстовых сообщений с типом 1 и одно сообщение нулевой длины с типом 255 второй программе. Вторая программа в цикле принимает сообщения любого типа в порядке FIFO и печатает их содержимое до тех пор, пока не получит сообщение с типом 255. Сообщение с типом 255 служит для неё сигналом к завершению работы и ликвидации очереди сообщений. Если перед запуском любой из программ очередь сообщений ещё отсутствовала в системе, то программа создаст её.

Запускать программы нужно с разных терминалов. Если первой запустить программу **ex1a**, а затем программу **ex1b**, то всё отработает нормально. Если сначала запустить программу **ex1b**, а затем **ex1a**, то первая программа заблокируется, пока не будет запущена вторая программа.

Обратите внимание на использование сообщения с типом 255 в качестве сигнала прекращения работы второго процесса. Это сообщение имеет нулевую длину, так как его информативность исчерпывается самим фактом наличия сообщения.

В описании системных вызовов *msgsnd()* и *msgrcv()* говорится о том, что передаваемая информации не обязательно должна представлять собой текст. Вы можете воспользоваться очередями сообщений для передачи данных любого вида. При передаче разнородной информации целесообразно информативную часть объединять внутри сообщения в отдельную структуру:

```
struct mymsgbuf {

  long mtype;

  struct {

    short sinfo;

    float finfo;

  } info;

} mybuf;
```

для правильного вычисления длины информативной части. В некоторых вычислительных системах числовые данные размещаются в памяти с выравниванием на определённые адреса (например, на адреса, кратные 4).

Поэтому реальный размер памяти, необходимой для размещения нескольких числовых данных, может оказаться больше суммы длин этих данных, т.е. в нашем случае

```
sizeof(info) >= sizeof(short) + sizeof(float)
```

Наличие у сообщений типов позволяет организовать двустороннюю связь между процессами через одну и ту же очередь сообщений. Процесс 1 может посылать процессу 2 сообщения с типом 1, а получать от него сообщения с типом 2. При этом для выборки сообщений в обоих процессах следует пользоваться вторым способом выбора.

Понятие мультиплексирования. Мультиплексирование сообщений. Модель взаимодействия процессов клиент — сервер. Неравноправность клиента и сервера

Используя технику из предыдущего примера, вы можете организовать получение сообщений одним процессом от множества других процессов через одну очередь сообщений и отправку им ответов через ту же очередь сообщений, т.е. осуществить *мультиплексирование* сообщений. Вообще под мультиплексированием информации понимают возможность одновременного обмена информацией с несколькими партнёрами. Метод мультиплексирования широко применяется в модели взаимодействия процессов *клиент–сервер*. В этой модели один из процессов является сервером. Сервер получает запросы от других процессов – клиентов – на выполнение некоторых действий и отправляет им результаты обработки запросов. Чаще всего модель клиент–сервер используется при разработке сетевых приложений, с которыми вы столкнётесь в материалах завершающих семинаров курса. Она изначально предполагает, что взаимодействующие процессы неравноправны:

1. Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически.
2. Сервер ждёт запроса от клиентов, инициатором же взаимодействия является клиент.
3. Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступать запросы от нескольких клиентов.

4. Клиент должен знать, как обратиться к серверу (например, какого типа сообщения он воспринимает) перед началом организации запроса к серверу, в то время как сервер может получить недостающую информацию о клиенте из пришедшего запроса.

Рассмотрим следующую схему мультиплексирования сообщений через одну очередь сообщений для модели клиент–сервер. Пусть сервер получает из очереди сообщений только сообщения с типом 1. В состав сообщений с типом 1, посылаемых серверу, процессы-клиенты включают значения своих идентификаторов процесса. Приняв сообщение с типом 1, сервер анализирует его содержание, выявляет идентификатор процесса, пославшего запрос, и отвечает клиенту, посылая сообщение с типом, равным идентификатору запрашивавшего процесса. Процесс-клиент после отправления запроса ожидает ответа в виде сообщения с типом, равным своему идентификатору. Поскольку идентификаторы процессов в системе различны, и ни один пользовательский процесс не может иметь PID равный 1, все сообщения могут быть прочитаны только теми процессами, которым они адресованы. Если обработка запроса занимает продолжительное время, сервер может организовывать параллельную обработку запросов, порождая для каждого запроса новый процесс-ребенок или новую нить исполнения.

Использование очередей сообщений для синхронизации работы процессов

В лекции 6 была доказана эквивалентность очередей сообщений и семафоров в системах, где процессы могут использовать разделяемую память. В частности, было показано, как реализовать семафоры с помощью очередей сообщений. Для этого вводился специальный синхронизирующий процесс-сервер, обслуживающий переменные-счётчики для каждого семафора. Процессы-клиенты для выполнения операции над семафором посылали процессу-серверу запросы на выполнение операции и ожидали ответа для продолжения работы. Теперь вы знаете, как это можно сделать в операционной системе UNIX и как, следовательно, можно использовать очереди сообщений для организации взаимоисключений и взаимной синхронизации процессов.