Семинар 7. Средства System V IPC. Семафоры и сообщения.

Семафоры. Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал Дейкстра (Dijkstra) в 1965 году.

Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen — проверять) и V (от verhogen — увеличивать). Классическое определение этих операций выглядит следующим образом:

P(S): пока S == 0 процесс блокируется;

S = S – 1;

V(S): S = S + 1;

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1.

Подобные переменные-семафоры могут быть с успехом применены для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях применяются через использование системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

Семафоры в UNIX. Отличие операций над UNIX семафорами от классических операций. При разработке средств System V IPC семафоры вошли в их состав как неотъемлемая часть. Следует отметить, что набор операций над семафорами System V IPC отличается от классического набора операций {P, V}, предложенного Дейкстрой. Он насчитывает три операции:

A(S, n) - увеличить значение семафора S на величину n;

D(S, n) - пока значение семафора S < n процесс блокируется. Далее S = S - n;

Z(S) - процесс блокируется до тех пор, пока значение семафора S не станет равным 0.

Изначально все IPC семафоры инициируются нулевым значением.

Легко видеть, что классической операции P(S) соответствует операция D(S,1), а классической операции V(S) соответствует операция A(S,1). Аналогом ненулевой инициализации Дейкстровских семафоров значением n может служить выполнение операции A(S,n) сразу после создания семафора S, с обеспечением атомарности создания семафора и ее выполнения посредством другого семафора. Мы показали, что классические семафоры реализуются через семафоры System V IPC. Обратное не является верным. Используя операции P(S) и V(S), мы не сумеем реализовать операцию Z(S).

Поскольку IPC семафоры являются составной частью средств System V IPC, то для них верно все, что говорилось об этих средствах в целом на предыдущем семинаре. IPC семафоры являются средством связи с непрямой адресацией, требуют инициализации для организации взаимодействия процессов и специальных действия для освобождения системных ресурсов по его окончании. Пространством имен IPC семафоров является множество значений ключа, генерируемых с помощью функции *ftok()*. Для совершения операций над семафорами системным вызовам в качестве параметра передаются IPC дескрипторы семафоров, однозначно идентифицирующих их во всей вычислительной системе, а вся информация о семафорах располагается в адресном пространстве ядра операционной системы. Это позволяет организовывать через семафоры взаимодействие процессов, даже не находящихся в системе одновременно.

Создание массива семафоров или доступ к уже существующему. Системный вызов *semget().* В целях экономии системных ресурсов операционная система UNIX позволяет создавать не по одному семафору для каждого конкретного значения ключа, а связывать с ключом целый массив семафоров (в Linux - до 500 семафоров в массиве, хотя это количество может быть уменьшено системным администратором). Для создания массива семафоров, ассоциированного с определенным ключом, или доступа по ключу к уже существующему массиву используется системный вызов *semget()*, являющийся аналогом системного вызова *shmget()* для разделяемой памяти, который возвращает значение IPC дескриптора для этого массива. При этом существуют те же способы создания и доступа, что и для разделяемой памяти. Вновь созданные семафоры инициируются нулевым значением.

Выполнение операций над семафорами. Системный вызов *semop().* Для выполнения операций A, D и Z над семафорами из массива используется системный вызов *semop()*, обладающий довольно сложной семантикой. Разработчики System V IPC явно перегрузили этот вызов, применяя его не только для выполнения всех трех операций, но

еще и для нескольких семафоров в массиве IPC семафоров одновременно. Для правильного использования этого вызова необходимо выполнить следующие действия:

1) Определиться, для каких семафоров из массива вы хотите выполнить операции. Необходимо иметь в виду, что все операции реально совершаются только перед успешным возвращением из системного вызова, т.е. если вы хотите выполнить операции A(S1,5) и Z(S2) в одном вызове и оказалось, что S2 != 0, то значение семафора S1 не будет изменено до тех пор, пока значение S2 не станет равным 0. Порядок выполнения операций в случае, когда процесс не переходит в состояние ожидание не определен. Так, например, при одновременном выполнении операций A(S1,1) и D(S2,1) в случае S2 > 1 неизвестно, что выполнится раньше - уменьшение значения семафора S2 или увеличение значения семафора S1. Если порядок является для вас существенным, лучше применить несколько вызовов вместо одного.

2) После того как вы определились с количеством семафоров и совершаемыми операциями, необходимо завести в программе массив из элементов типа *struct sembuf* с размерностью равной определенному количеству семафоров (если операция совершается только над одним семафором можно, естественно, обойтись просто переменной). Каждый элемент этого массива будет соответствовать операции над одним семафором.

3) Заполнить элементы массива. В поле *sem_flg* каждого элемента занести значение 0 (с другими значениями флагов мы на семинарах работать не будем). В поля *sem_num* и *sem_op* занести номера семафоров в массиве IPC семафоров и соответствующие коды операций. Семафоры нумеруются, начиная с 0. Если у вас в массиве всего один семафор, то он будет иметь номер 0. Операции кодируются так:

– для выполнения операции A(S,n) значение поля *sem_op* должно быть равно n.

– для выполнения операции D(S,n) значение поля *sem_op* должно быть равно -n.

– для выполнения операции Z(S) значение поля *sem_op* должно быть равно 0.

4) В качестве второго параметра системного вызова *semop()* указать адрес заполненного массива, а в качестве третьего параметра - ранее определенное количество семафоров, над которыми совершаются операции.

Задания: 1) Написать 2 программы, первая из которых завершает работу только после запуска второй.

2) Написать 2 программы, первая программа ждет 5 запусков второй, после чего работает без блокировки.

Удаление набора семафоров из системы с помощью команды *ipcrm* или системного вызова *semctl().* Как мы говорили и видели из примеров, массив семафоров может продолжать существовать в системе и после завершения использовавших его

процессов, а семафоры будут сохранять свое значение. Это способно привести к неправильному поведению программ, предполагающих, что семафоры были только что созданы и, следовательно, имеют нулевое значение. Необходимо удалять семафоры из системы перед запуском таких программ или перед их завершением. Для удаления семафоров можно воспользоваться командами *ipcs* и *ipcrm*, рассмотренными на предыдущем семинаре. Команда *ipcrm* в этом случае должна иметь вид

ipcrm sem <IPC идентификатор>

Для этой же цели мы можем применять системный вызов *semctl()*, который умеет выполнять и другие операции над массивом семафоров, но их рассмотрение выходит за рамки нашего курса.

Очереди сообщений в UNIX как составная часть System V IPC. Так как очереди сообщений входят в состав средств System V IPC, то для них верно все, что говорилось ранее об этих средствах в целом и уже знакомо нам. Очереди сообщений, как и семафоры, как 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 имен очередей сообщений является то же самое множество значений ключа, генерируемых с помощью функции *ftok()*. Для выполнения примитивов *send* и *receive* 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, хранимой в каждой очереди. Каждый элемент списка представляет собой отдельное сообщение. Сообщения имеют атрибут, называемый типом сообщения. 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) Первым выбирается сообщение с минимальным типом, не превышающим некоторого заданного значения, пришедшее ранее всех других сообщений с тем же типом.

Реализация примитивов *send* и *receive* обеспечивает скрытое от пользователя взаимоисключение во время помещения сообщения в очередь или его получении из очереди, а также 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 организовывать 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 определенным 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.

Реализация примитивов *send* и *receive*. Системные вызовы *msgsnd()* и *msgrcv()*. Для выполнения примитива *send* служит системный вызов *msgsnd()*, копирующий пользовательское сообщение в очередь сообщений, заданную своим IPC дескриптором. При изучении описания этого вызова обратите особое внимание на следующие моменты:

Тип данных *struct msgbuf* 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 обязана 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 наличия сообщения (сообщение используется как сигнальное средство связи).

На наших занятиях мы, как правило, будем использовать нулевое значение флага системного вызова, которое приводит к блокировке процесса при отсутствии достаточного свободного места в очереди сообщений.

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

Тип данных *struct msgbuf*, как и для вызова *msgsnd()*, является лишь шаблоном для пользовательского типа данных.

Способ выбора сообщения задается нулевым, положительным или отрицательным значением параметра *type*. Точное значение типа выбранного сообщения можно

определить из соответствующего поля структуры, 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*.

На наших занятиях мы будем, как правило, пользоваться нулевым значением флагов 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 информативной части сообщения в операционной системе Linux составляет 4080 байт и может быть уменьшена при генерации системы. Текущее значение максимальной длины 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 нам 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 их рассмотрение лежит вне нашего курса. 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.

Задания: 1) Написать 2 программа, первая передает 5 сообщений второй, после чего передает сигнал завершения.

2) Модифицировать для передачи чисел.

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

Используя технику из предыдущего примера, мы можем 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 партнерами. 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. Наиболее часто модель клиент-сервер используется при разработке сетевых приложений, с которыми мы столкнемся на завершающих семинарах курса. Она изначально предполагает неравноправность взаимодействующих процессов:

Сервер, как правило, работает постоянно, на всем протяжении жизни приложения, а клиенты могут работать эпизодически. Сервер ждет запроса от клиентов, инициатором же взаимодействия выступает клиент. Как правило, клиент обращается к одному серверу за раз, в то время как к серверу могут одновременно поступить запросы от нескольких клиентов. 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 значение 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 посылки 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.