Архитектура ЭВМ. Операционные системы

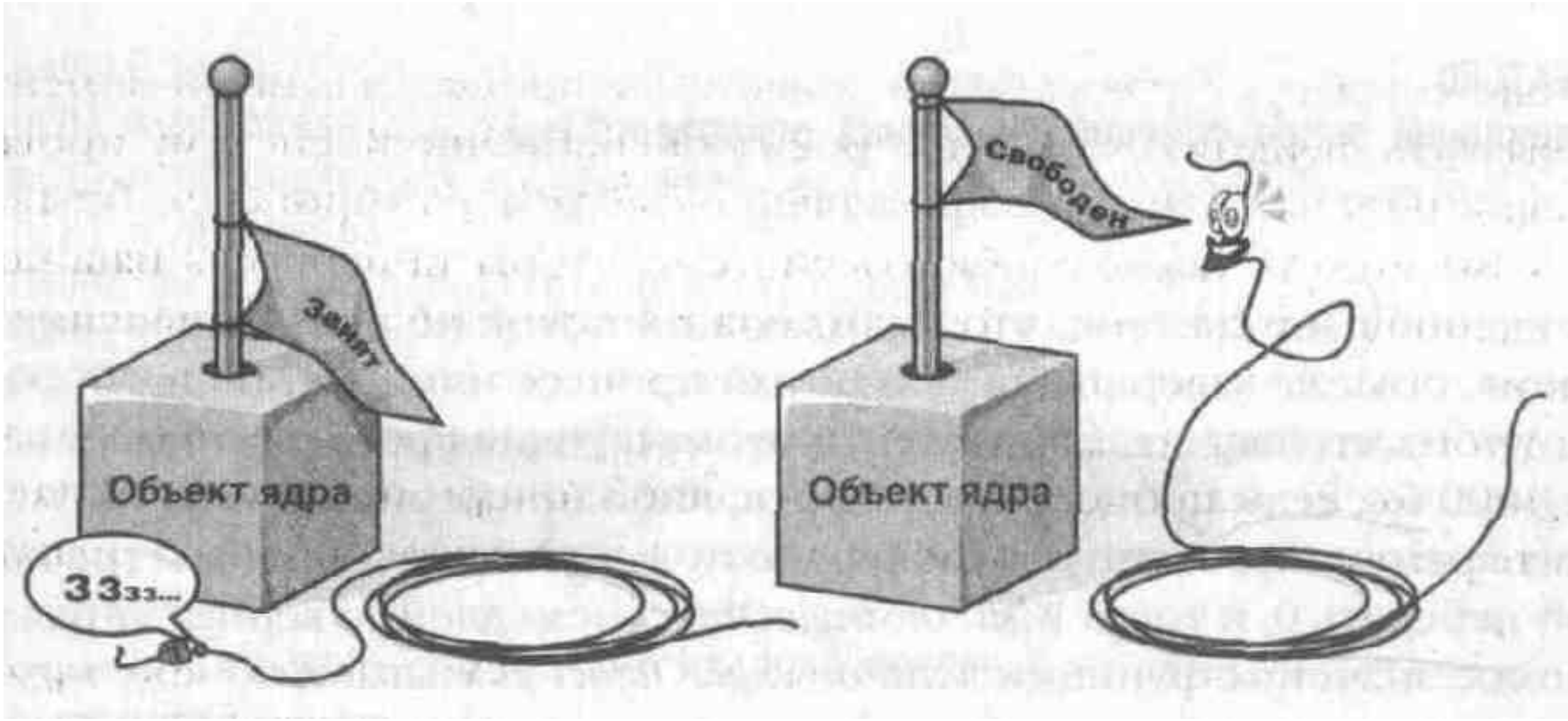
Власов Е.Е.

Синхронизация обмена данными

Для корректного обмена данными необходима в случае, когда два и более выполняющихся процесса (потока) могут одновременно читать и писать данные.
Обычно для этого используются исключающие блокировки – механизмы, при которых в один момент времени доступ к ресурсу для записи и чтения имеет только один процесс, а остальные находятся в состоянии ожидания.

Когда поток изменяет значение переменной, существует потенциальная опасность, что другой поток может прочитать еще не до конца записанное значение. На аппаратных платформах, где запись в память осуществляется более чем за один цикл, может произойти так, что между двумя циклами записи вклинится цикл чтения. Разумеется, такое поведение во многом зависит от аппаратной архитектуры, но при написании переносимых программ мы не можем полагаться на то, что они будут выполняться только на определенной платформе.


В ГОВНО НАСТУПИЛ!
СИНХРОНИЗАЦИЯ ПОТОКОВ
risovach.ru


Занят
Объект ядра
ЗЗзз...
Свободен
Объект ядра

Объекты синхронизации

- Семафоры (semaphore)
- Мьютексы (Mutaual Exception)
- Взаимные блокировки (spinlock)
- Блокировки чтения-записи
- Условные переменные
- Атомарные операции
- Сериальные блокировки
- Переменные, локальные для каждого процессора
- Семафоры для чтения/записи
- Мьютексы реального времени
- Механизмы ожидания завершения

Семафоры

Механизм семафоров, реализованный в ОС UNIX, является обобщением классического механизма семафоров общего вида, предложенного Дейкстрой.

Семафор в ОС UNIX состоит из следующих элементов:

- значение семафора;
- идентификатор процесса, который хронологически последним работал с семафором;
- число процессов, ожидающих увеличения значения семафора;
- число процессов, ожидающих нулевого значения семафора.

Для работы с семафорами поддерживаются три системных вызова:

- `semget` для создания и получения доступа к набору семафоров;
- `semop` для манипулирования значениями семафоров (это именно тот системный вызов, который позволяет процессам синхронизоваться на основе использования семафоров);
- `semctl` для выполнения разнообразных управляющих операций над набором семафоров.

Создание семафора

```
int semget(key_t key, int count, int flag);
```

прямые параметры `key` и `flag` и возвращаемое значение системного вызова имеют тот же смысл, что для других системных вызовов семейства "get", а параметр count задает число семафоров в наборе семафоров, обладающих одним и тем же ключом. После этого индивидуальный семафор идентифицируется дескриптором набора семафоров и номером семафора в этом наборе. Если к моменту выполнения системного вызова `semget` набор семафоров с указанным ключом уже существует, то обращающийся процесс получит соответствующий дескриптор, но так и не узнает о реальном числе семафоров в группе.

Управление состоянием семафора

```
oldval = semop(int semid, struct sembuf *oplist,
                    size_t count);
```

где `id` - это ранее полученный дескриптор группы семафоров, oplist - массив описателей операций над семафорами группы, а `count` - размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива `oplist` имеет следующую структуру:

```
 struct sembuf {
   u_short sem_num;        /* semaphore # */
   short   sem_op;         /* semaphore operation */
   short    sem_flg;       /* operation flags */
};
```

Если проверка прав доступа проходит нормально, и указанные в массиве oplist номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов выполняется следующим образом. Для каждого элемента массива oplist значение соответствующего семафора изменяется в соответствии со значением поля "операция".

- Если значение поля операции положительно, то значение семафора увеличивается на значение поля sem_op, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии UNIX).
- Если значение поля операции равно нулю, то если значение семафора также равно нулю, выбирается следующий элемент массива oplist. Если же значение семафора отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания (усыпляется в терминологии UNIX).
- Если значение поля операции отрицательно, и его абсолютное значение меньше или равно значению семафора, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует (пробуждает) все процессы, ожидающие нулевого значения этого семафора. Если же значение семафора меньше абсолютной величины поля операции, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и откладывает (усыпляет) текущий процесс до наступления этого события.

Основным поводом для введения массовых операций над семафорами было стремление дать программистам возможность избегать тупиковых ситуаций в связи с семафорной синхронизацией. Это обеспечивается тем, что системный вызов `semop`, каким бы длинным он не был (по причине потенциально неограниченной длины массива `oplist`) выполняется как атомарная операция, т.е. во время выполнения `semop` ни один другой процесс не может изменить значение какого-либо семафора.

Среди флагов-параметров системного вызова `semop` может содержаться флаг с символическим именем `IPC_NOWAIT`, наличие которого заставляет ядро ОС UNIX не блокировать текущий процесс, а лишь сообщать в ответных параметрах о возникновении ситуации, приведшей бы к блокированию процесса при отсутствии флага `IPC_NOWAIT`.

Управление объектами семафоров

```
int semctl(int semid, int number, int cmd, arg);
```

Функция **semctl** позволяет выполнять операции, определенные в *cmd* над набором семафоров, указанным в *semid* или над семафором с номером *semnum* из этого набора. (Семафоры нумеруются, начиная с 0.) Функция имеет три или четыре аргумента. Если аргументов четыре, то вызов выглядит как **semctl(***semid***,***semnum***,***cmd***,***arg***);** где четвертый аргумент *arg* имеет тип **union semun**, определенный как

```
union semun {
   int val; /* value for SETVAL */
   struct semid_ds *buf; /* buffer for IPC_STAT, IPC_SET */
   unsigned short *array; /* array for GETALL, SETALL */
    /* Linux specific part: */
   struct seminfo *__buf; /* buffer for IPC_INFO */
};
```

Аргумент *cmd* может принимать следующие значения:

- **IPC_STAT** Скопируйте информацию из структуры данных набора семафоров в структуру, указанную в *arg***.buf**. Аргумент *semnum* игнорируется. Вызывающий процесс должен прочитать привилегии доступа в наборе семафоров.
- **IPC_SET** Внесите значения некоторых членов структуры **semid_ds**, на которую указывает *arg***.buf** , в структуру данных набора семафоров и обновите **sem_ctime.** Присвоить следующим полям структуры данных **struct semid_ds** соответствующие значения, на которые указывает *arg***.buf sem_perm.uid sem_perm.gid sem_perm.mode** /* Только младшие 9 битов */ Эта команда может выполняться только процессом, который имеет действующий идентификатор пользователя, равный либо идентификатору суперпользователя, либо создателя или владельца набора семафоров. Аргумент *semnum* игнорируется.
- **IPC_RMID** Немедленно удалить из системы набор семафоров и структуры его данных, запускающие все процессы, находящиеся в режиме ожидания (при этом возвращается сообщение об ошибке, а **errno** присваивается значение **EIDRM**). Эта команда может выполняться только процессом, который имеет действующий идентификатор пользователя, равный либо идентификатору суперпользователя, либо создателя или владельца набора семафоров. Аргумент *semnum* игнорируется.
- **GETALL** Возвращает значение **semval** всем семафорам в массиве *arg***.array**. Аргумент *semnum* игнорируется. Для этого вызывающему процессу нужны права на чтение.
- **GETNCNT** Системный вызов возвращает значение **semncnt** семафору *semnum*-th (например, число процессов, ожидающих увеличения значения **semval** семафора *semnum*-th). Для этого вызывающему процессу нужны права на чтение.
- **GETPID** Системный вызов возвращает значение **sempid** семафору *semnum*-th (например, идентификатор процесса, который последним делал вызов **semop** семафору *semnum*-th). Для этого вызывающему процессу нужны права на чтение.
- **GETVAL** системный вызов возвращает значение **semval** семафору *semnum*-th. Для этого вызывающему процессу нужны права на чтение.
- **GETZCNT** Системный вызов возвращает значение **semzcnt** семафору *semnum*-th (например, количество процессов, ожидающих, чтобы значение **semval** семафора *semnum*-th стало равным нулю). Для этого вызывающему процессу нужны права на чтение.
- **SETALL** Установить значение **semval** всех семафоров равным значениям элементов массива, на который указывает *arg***.array**, изменяя также **sem_ctime**, являющееся членом структуры **semid_ds ;** а эта структура ассоциируется с набором семафоров. История отменяемых операций удаляется для всех измененных семафоров во всех процессах. Процессы, находящиеся в очереди, активизируются, если **semval** становится равным нулю или значение его увеличивается. Аргумент *semnum* игнорируется. Для этого вызывающему процессу нужны права на чтение.
- **SETVAL** Установите значение **semval** на указанное в *arg***.val** для всех семафоров *semnum*-th, изменяя также **sem_ctime** в структуре **semid_ds**, соотносимой с набором семафоров. История отмененных операций удаляется для всех измененных семафоров во всех процессах. Процессы, находящиеся в очереди, активизируются, если **semval** становится равным нулю

Мьютексы

Мьютексы можно рассматривать как двоичные семафоры.

блокировка, которая устанавливается (запирается) перед обращением к разделяемому ресурсу и снимается (отпирается) после выполнения требуемой последовательности операций. Если мьютекс заперт, то любой другой поток, который попытается запереть его, будет заблокирован до тех пор, пока мьютекс не будет отперт.

Если в момент, когда отпирается мьютекс, заблокированными окажутся несколько потоков, все они будут запущены и первый из них, который успеет запереть мьютекс, продолжит работу. Все остальные потоки обнаружат, что мьютекс по-прежнему заперт, и опять перейдут в режим ожидания. Таким образом, доступ к ресурсу сможет получить одновременно только один поток.

Такой механизм взаимоисключений будет корректно работать только при условии, что все потоки приложения будут соблюдать одни и те же правила доступа к данным. Операционная система никак не упорядочивает доступ к данным. Если мы позволим одному потоку производить действия с разделяемыми данными, предварительно не ограничив доступ к ним, то остальные потоки могут обнаружить эти данные в противоречивом состоянии, даже если перед обращением к ним будут устанавливать блокировку.

Переменные-мьютексы определяются с типом `pthread_mutex_t`. Прежде чем использовать переменную мьютекс, мы должны сначала инициализировать ее, записав в нее значение константы `PTHREAD_MUTEX_INITIALIZER` (только для статически размещаемых мьютексов) или вызвав функцию `pthread_mutex_init`.

Системные вызовы работы с mutex

- pthread_mutex_init
- pthread_mutex_destroy
- pthread_mutex_lock
- pthread_mutex_trylock
- pthread_mutex_unlock

Системные вызовы pthread_mutex_init и int pthread_mutex_destroy

```
#include <pthread.h>
int pthread_mutex_init(
   pthread_mutex_t *restrict mutex,
   const pthread_mutexattr_t *restrict attr);
```

Инициализирует мьютекс переданный параметром `mutex`. Для того, чтобы использовать параметры по умолчанию необходимо вместо attr передать NULL.

```
int pthread_mutex_destroy(pthread_mutex_t *restrict
mutex);
```

Уничтожает `mutex`.

Системный вызов pthread_mutex_lock

#include <pthread.h>

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Блокирует мьютекс. Если мьюеткс уже заблокирован, то вызывающий поток блокируется.

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Если к моменту вызова этой функции мьютекс будет отперт, функция запрет мьютекс и вернет значение 0. В противном случае pthread_mutex_trylock вернет код ошибки EBUSY. Вызывающий поток при в этом случае не плокируется.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```