POSIX Threads

Семафор – это объект, который используется для контроля доступа нескольких потоков до общего ресурса. В общем случае это переменная, состояние которой изменяется каждым из потоков. Текущее состояние переменной определяет доступ к ресурсам.

В `pthreads` семафор – это переменная типа `sem_t`, которая может находиться в заданном числе состояний. Каждый поток может увеличить счётчик семафора, или уменьшить его. Операция увеличения значения счётчика называется `V – vacate` (освобождать). Уменьшение счётчика – это операция `P procure` – добывать. Операции `P` и `V` выполняются атомарно.

POSIX Threads

При вызове потоком **P**(sem) если семафор "свободен" (>0), его значение уменьшается на 1, и выполнение потока продолжается, если семафор "занят" (<=0), поток переводится в состояние ожидания и помещается в очередь, соответствующую данному семафору запускается какой-либо другой готовый к выполнению поток при вызове потоком **V**(sem) если очередь потоков, ассоциированная с данным семафором, не пуста – один из них разблокируется и помещается в очередь готовых к выполнению поток, вызвавший V (sem), продолжает свое выполнение в противном случае (нет потоков, ожидающих освобождения семафора), значение семафора увеличивается. Все вызовы изменяют состояние семафора, то есть семафор сохраняет некоторую информацию о прошедших вызовах.

POSIX Threads

Семафоры описаны в заголовочном файле `semaphore.h`.
Именованные семафоры: они могут быть использованы для синхронизации между несколькими несвязанными процессами.
Безымянные семафоры: они могут быть использованы потоками внутри процесса или для синхронизации между связанными процессами (например, родительским и дочерним процессом).
Безымянный семафор можно инициализировать с помощью процедуры:

```
int sem_init(sem_t *sem, int pshared, unsigned int
value)
```

где `sem` – это указатель на область памяти, где находится семафор, `pshared` – флаг, если равен 0, семафор используется потоками одного процесса, в противном случае доступ к нему могут иметь несколько процессов, `value -` значение, которым будет инициализирован семафор.

POSIX Threads

В случае если не именованный семафор больше не требуется его необходимо утилизировать с помощью процедуры:

```
int sem_destroy(sem_t *sem)
```

Процедуру уничтожения обычно вызывают после исполнения после `pthread_join()`. Результат действий с использованием уничтоженного семафора не определен.

POSIX Threads

Процедура

```
sem_t* sem_open(const char *name, int oflag,…
 /* mode_t mode, unsigned int value */)
```

Создаёт новый именованный семафор или открывает существующий. Семафору присваивается имя `name`. Возвращаемое значение представляет собой указатель на тип `sem_t`. При ошибке процедура возвращает нулевой указатель и устанавливает `errno`. Если процесс попытается несколько раз открыть один и тот же семафор, ему будут возвращать один и тот же указатель. Именованные семафоры всегда разделяемые между процессами. При доступе к существующему семафору проверяются права доступа по той же схеме, по которой в `Unix` системам проверяются права доступа к файлам. Для доступа к семафору процесс должен иметь права чтения и записи.

POSIX Threads

`name` - имя семафора. Имя должно начинаться с символа `'/'` и не должно содержать других символов `'/'`. Рекомендуется, чтобы имя не превышало `14` символов. В зависимости от реализации, объект с таким именем может либо появляться либо не появляться в корневом каталоге корневой файловой системы. Для
создания семафора не обязательно иметь право создания файлов в корневом каталоге.
`flags` - флаги. Может принимать значения `0`, `O_CREAT` и `O_CREAT|O_EXCL`, где `O_CREAT` и `O_EXCL` - константы, определенные в `<sys/fcntl.h>`. Смысл этих значений аналогичен соответствующим значениям флагов в параметрах open. 0 означает попытку доступа к уже существующему семафору, `O_CREAT` - доступ к существующему семафору или попытку создания, если такого семафора нет, `O_EXCL` - ошибку, если при попытке создания обнаруживается, что такой семафор уже существует.
`mode` - необязательный параметр, который используется, только если flags содержит бит `O_CREAT`. Обозначает права доступа к семафору, которые задаются девятибитовой маской доступа, похожей на маску доступа к файлам. Как и у файла, у семафора есть идентификаторы хозяина и группы. Идентификатор хозяина устанавливается равным эффективному \
идентификатору пользователя процесса, создавшего семафор, идентификатор группы - эффективному идентификатору группы процесса.
`value` - необязательный параметр, который используется только если `flags` содержит бит `O_CREAT`. Содержит начальное значение флаговой переменной семафора при его создании. Не может превышать константу `SEM_VALUE_MAX.`

POSIX Threads

Открыв семафор с помощью `sem_open`, можно потом закрыть его с помощью процедуры

```
int sem_close(sem_t *sem)
```

Операция закрытия выполняется автоматически при завершении процесса для всех семафоров, которые были им открыты. Автоматическое закрытие осуществляется как при добровольном завершении работы, так и при принудительном (с помощью сигнала).

POSIX Threads

Закрытие семафора не удаляет его из системы. Значение семафора сохраняется, даже если ни один процесс не держит его открытым. Именованный семафор удаляется из системы вызовом `sem_unlink`:

```
int sem_unlink(const char *name)
```

Для каждого семафора ведется подсчет процессов, в которых он является открытым (как и для файлов), и функция `sem_unlink` действует аналогично `unlink` для файлов: объект `name` может быть удален из файловой системы, даже если он открыт какими-либо процессами, но реальное удаление семафора не будет осуществлено до тех пор, пока он не будет окончательно закрыт. Процедура возвращает ноль в случае успешного завершения и минус один в случае ошибки. Исполнять `sem_unlink` могут только владелец семафора и суперпользователь.

POSIX Threads

Для управления семафорами используют процедуры:

`int sem_post(sem_t *sem)` - атомарно увеличивает значение семафора на единицу.

`int sem_wait(sem_t *sem)` - если значение семафора больше нуля, то оно уменьшается на единицу и нить продолжает работать. В противном случае нить переводится в состояние ожидания, потенциально бесконечного. Нить пробуждается, когда какая-либо другая нить выполнит sem_post для данного семафора.

POSIX Threads

Если блокировка нити в случае нулевого значения семафора нежелательна, то можно использовать процедуру :

```
int sem_trywait(sem_t *sem)
```

Функция ожидания с проверкой семафора, до выполнения операции над семафором проверяет значение счетчика. Если это значение больше нуля, функция выполняет декремент счетчика, а если значение равно нулю - возвращает управление нити, не блокируясь на ожидании доступности семафора (но захват семафора в этом случае не происходит, о чем извещает код возврата).

POSIX Threads

Процедура ожидания с тайм-аутом:

```
int sem_timedwait(sem_t* sem, const
struct timespec * abs_timeout)
```

Ожидает возможности уменьшить на `1` счетчик семафора до момента времени `abs_timeout`.

POSIX Threads

При успешном выполнении все функции возвращают ноль. При ошибке значение семафора не изменяется, возвращается -1, а в `errno` указывается причина ошибки.
Возможные ошибки:
`EINTR` - вызов был прерван обработчиком сигнала.
`EINVAL` - значение `sem` не является корректным для семафора.
В `sem_trywait()` может возникать следующая дополнительная ошибка:
`EAGAIN` - операция не может быть выполнена без блокировки (т. е., значение семафор равно нулю).
В `sem_timedwait()` дополнительно могут возникать следующие ошибки:
`EINVAL` - значение `abs_timeout.tv_nsecs` меньше `0`, или больше или равно `1000` миллионов.
`ETIMEDOUT` - истёк период ожидания в вызове раньше возможности блокировки семафора.

POSIX Threads

Процедура int sem_getvalue(sem_t *sem, int *sval)
Помещает текущее значение семафора, заданного в `sem`, в виде целого, на которое указывает `sval`. При успешном выполнении `sem_getvalue()` возвращается ноль, при ошибке возвращается `-1`, а в `errno` содержится код ошибки.

POSIX Threads

```
sem_t sem;
void * thread_func(void *arg)
{
  int i;
  int id = * (int *) arg;
  sem_post(&sem);
  for (i = 0; i < 4; i++) {
    printf("Hello from thread %d\n", id);
    sleep(1);
  }
}
int main(int argc, char * argv[])
{
  int id=1;
  int result;
  pthread_t thread1, thread2;
  sem_init(&sem, 0, 0);
  result = pthread_create(&thread1, NULL, thread_func, &id);
  if (result != 0){ perror(«While creating thread»); return 1;}
  sem_wait(&sem);
  id = 2;
  result = pthread_create(&thread2, NULL, thread_func, &id);
  if (result != 0){ perror(«While creating thread»); return 1; }
  result = pthread_join(thread1, NULL);
  if (result != 0){ perror(«While joining thread»); return 2; }
  result = pthread_join(thread2, NULL);
  if (result != 0){ perror(«While joining thread»); return 2; }
  sem_destroy(&sem);
  return 0;
}
```

POSIX Threads

Решение задачи производитель-потребитель на двух семафорах.

```
sem_t p, q;
int data;
void *producer(void *) {
  while(1) {
    int t=produce();
    sem_wait(&p);
    data=t;
    sem_post(&q);
  }
  return NULL;
}

void *consumer(void *) {
  while (1) {
    int t;
    sem_wait(&q);
    t=data;
    sem_post(&p);
    consume(t);
  }
  return NULL;
}
```

POSIX Threads

`RW-lock` (блокировка чтения-записи, блокировка «много читателей, один писатель») – это примитив синхронизации, который позволяет решить проблему «читателей-писателя». Проблема возникает, когда используется много нитей читателей и один писатель, часто встречается при решении практических задач. Простой подход – использованием мьютекса – считается медленным, так как нет необходимости блокировать ресурс, когда его только читают. Проблема решается с помощью блокировки чтения-записи, которая обеспечивает множественный доступ потоков читателей, и эксклюзивный доступ потока писателя.

POSIX Threads

`RW-lock` имеет два набора функций блокировки-освобождения ресурса – один для потоков читателей, другой для потока писателя. Инициализируется ресурс типа `pthread_rwlock_t` либо стандартным инициализатором `PTHREAD_RWLOCK_INITIALIZER` , либо процедурой:

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock, const
pthread_rwlockattr_t *attr)
```

rwlock - указатель на блокировку, attr – атрибуты блокировки. Во время инициализации, могут появиться ошибки:
`EAGAIN` – у системы нет ресурсов для инициализации новой блокировки
`ENOMEM` – у системы не хватает памяти для инициализации ресурса
`EPERM` – вызывающая функция не обладает правами на создание блокировки
`EBUSY` – система обнаружила попытку повторной инициализации уже инициализированного, но не удалённого ресурса
`EINVAL` – неправильные атрибуты блокировки

POSIX Threads

Объект типа pthread_rwlockattr_t, содержащий в себе атрибуты может быть инициализирован с помощь процедуры:

```
int pthread_rwlockattr_init (
  pthread_rwlockattr_t *attr)
```

После использования объект можно уничтожить с помощью процедуры:

```
int pthread_rwlockattr_destroy (
  pthread_rwlockattr_t *attr)
```

POSIX Threads

После инициализации объекта типа `pthread_rwlockattr_t` для установки или сброса отдельных атрибутов используются специальные функции. Единственный определенный на настоящее время атрибут - `PTHREAD_PROCESS_SHARED`, который указывает на то, что блокировка используется несколькими процессами, а не отдельными потоками одного процесса. Для управления этим флагом существует процедура `pthread_rwlockattr_setpshared`, которая устанавливает значение этого атрибута равным `value`, которое может быть либо `PTHREAD_PROCESS_PRIVATE`, либо `PTHREAD_PROCESS_SHARED`.

```
int
pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, i
nt value )
```

Для взятия параметра существует процедура:

```
int pthread_rwlockattr_getpshared(const
pthread_rwlockattr_t *attr, int *valptr)
```

POSIX Threads

Уничтожение блокировки производится с помощью процедуры:

```
int
pthread_rwlock_destroy(pthread_rwlock_t
*rwlock)
```

В случае успешного выполнения возвращает ноль, иначе код ошибки.
`EBUSY` – попытка уничтожения захваченной блокировки
`EINVAL` – неправильные аргументы процедуры

POSIX Threads

Блокировка разрешает всем потокам доступ по чтению к
некоторой области данных с помощью вызова:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
```

Если блокировка захвачена для записи, то вызывающая нить не получит блокировку для чтения. Процедура `pthread_rwlock_tryrdlock`, как и процедура `pthread_rwlock_rdlock`, устанавливает блокировку для чтения. Однако в тех случаях, когда объект `rwlock` захвачен какой-либо нитью для записи, или есть нити, ожидающие получения блокировки `rwlock` для записи, эта функция возвращает сообщение об ошибке. Нить может несколько раз захватить блокировку `rwlock` для чтения (то есть вызвать функцию `pthread_rwlock_rdlock` *n* раз). В этом случае нить должна соответствующее число раз разблокировать объект (то есть вызвать функцию `pthread_rwlock_unlock` *n* раз).

```
int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock)
```

Процедура постарается захватить `rwlock` для чтения или будет ожидать, пока `rwlock` не освободят до наступления момента `abstime`.

```
int pthread_rwlock_timedrdlock (
pthread_rwlock_t * rwlock,
const struct timespec * abstime)
```

Вызов любой из этих функций для неинициализированной блокировки чтения/записи может привести к непредсказуемым результатам.

POSIX Threads

Установка блокировки на запись с помощью процедуры:

```
int pthread_rwlock_wrlock( pthread_rwlock_t *rwlock)
```

Когда поток, желающий выполнить запись, обращается с вызовом pthread_rwlock_wrlock, то требование записи блокируется до тех пор, пока все читающие потоки завершат чтение области данных. Множество пишущих потоков упорядочивается в очередь и ожидает возможности выполнить запись к защищенной структуре данных. Имеются также специальные системные вызовы позволяющие потокам проверить на допустимость выполнения действия без блокировки потока. Данная процедура захватывают rwlock, если он доступен.

```
int pthread_rwlock_trywrlock( pthread_rwlock_t *rwlock)
```

Процедура `pthread_rwlock_timedwrlock` применяет блокировку с ожиданием, если до момента времени `abstime` читатели не освободят ресурс, ожидание будет прервано.

```
int pthread_rwlock_timedwrlock( pthread_rwlock_t * rwlock, const
struct timespec * abstime)
```

Снятие блокировки чтение-запись:

```
int pthread_rwlock_unlock( pthread_rwlock_t *rwlock)
```

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
void compResults(char *string, int rc) {
  if (rc) {
    printf("Ошибка в : %s, rc=%d", string, rc);
    exit(EXIT_FAILURE);
  }
  return;
}
void *rdlockThread(void *arg)
{
  int rc;
  int count=0;
  printf("Выполняется нить, получение блокировки для чтения с ожиданием\n");
  do{
    rc = pthread_rwlock_tryrdlock(&rwlock);
    if(rc == EBUSY){
      if (++count >= 10) {
        printf("Достигнуто ограничение на число попыток, ошибка!\n");
        exit(EXIT_FAILURE);
      }
      printf("Не удалось получить блокировку, выполнение других операций, а затем повтор...\n");
      sleep(1);
    }else
      break;
  }while(1); compResults(«pthread_rwlock_tryrdlock() 1\n", rc);
    sleep(2);
    printf("освобождение блокировки для чтения\n");
    rc = pthread_rwlock_unlock(&rwlock); compResults("pthread_rwlock_unlock()\n", rc);
    return NULL;
  }
  compResults("pthread_rwlock_unlock()\n", rc);
  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);
  rc = pthread_rwlock_destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);
  return 0;
}
```

```
int main(int argc, char **argv)
{
  int                     rc=0;
  pthread_t thread;
  printf("Главная нить, получение блокировки для записи\n");
  rc = pthread_rwlock_wrlock(&rwlock);
  compResults("pthread_rwlock_wrlock()\n", rc);
  printf("Главная нить, создание нити для вызова функции
pthread_rwlock_tryrdlock()\n");
  rc = pthread_create(&thread, NULL, rdlockThread, NULL);
  compResults("pthread_create\n", rc);

  printf("Главная нить удерживает блокировку для записи\n");
  sleep(5);

  printf("Главная нить, освобождение блокировки для записи\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_rwlock_destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);
  return 0;
}
```