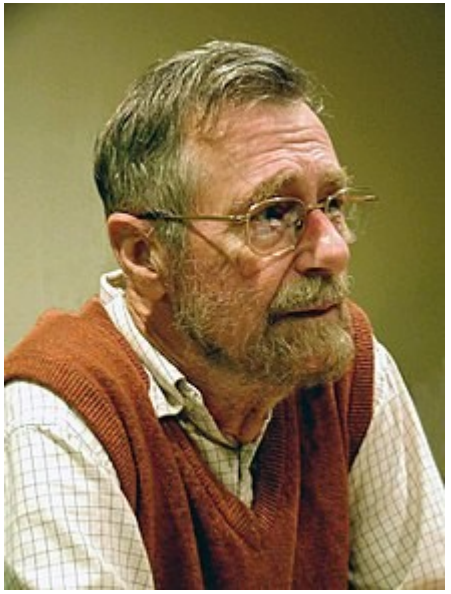
Синхронизация потоков - 2

- Семафоры
- Условные переменные

Sun Studio 12: Thread Analyzer User's Guide
https://docs.oracle.com/cd/E19205-01/820-0619/index.html

Семафор



Edsger Wybe Dijkstra
11 May 1930 - 6 August 2002

The semaphore concept
was invented by Edsger Wybe Dijkstra
in 1962 or 1963

Семафор (англ. semaphore) — примитив синхронизации потоков и процессов в основе которого лежит неотрицательное целое число — счетчик. С этим числом разрешается выполнять две атомарные операции: увеличение и уменьшение значения счетчика на единицу. Операция уменьшения для нулевого значения счетчика является блокирующей.

Часто используется для построения более сложных механизмов синхронизации и применяется для синхронизации параллельно работающих задач, для защиты передачи данных через разделяемую память, для защиты критических секций, а также для управления доступом к аппаратному обеспечению.

Операции уменьшения и увеличения значения семафора первоначально обозначались буквами **P** (от нидерл. *Proberen* — пытаться) и **V** (от нидерл. *verhogen* — поднимать выше) соответственно. Данные обозначения дал операциям над семафорами Дейкстра, но так как они не понятны для людей, говорящих на других языках, на практике обычно используются другие обозначения.

Для названия этих операций мы будем использовать обозначения операций: P и W.

W-операция (операция ожидания — wait) над семафором представляет собой попытку уменьшения значения семафора на 1. Если перед выполнением W-операции значение семафора было больше 0, W-операция выполняется без задержек. Если перед выполнением W-операции значение семафора было 0, поток, выполняющий W-операцию, переводится в состояние ожидания до тех пор, пока значение семафора не станет большим 0 (вследствие действий, выполняемых другими потоками) После снятия блокировки значение семафора уменьшается на единицу и операция завершается.

P-операция (операция установки — post) над семафором представляет собой увеличение значения семафора на 1. Если до этого счетчик был равен нулю и существовали потоки, заблокированные в операции ожидания данного семафора, один из этих процессов выходит из состояния ожидания и может выполнить свою W-операцию (т.е. счетчик семафора снова становится равным нулю).

Операция W аналогична операции захвата мьютекса, а операция P — операции его освобождения. Однако, в отличие от мьютексов, операции W и P не обязаны выполняться одним и тем же потоком исполнения и даже не обязаны быть парными. Это позволяет использовать семафоры в различных ситуациях, которые сложно или невозможно разрешить при помощи мьютексов.
Рассмотрим одну реализацию семафоров для потоков, соответствующую стандарту POSIX.

Этап 1. Определение переменной

```
#include <semaphore.h>
sem_t sem;
```

Этап 2. Инициализация

```
int sem_init(sem_t *sem,
   int pshared,              /* тип семафора */
   unsigned int value   /* начальное значение */);
```

pshared == 0 - семафор локален по отношению к текущему процессу

pshared != 0 - семафор может быть совместно использован разными процессами (вопрос по использованию).

Повторная инициализация уже инициализированного семафора приводит к неопределенному поведению

Результат:

= 0 — успешное завершение;

-1 — ошибочное завершение (для индикации ошибки используется переменная *errno*)

Возможные ошибки:

EINVAL — значение аргумента value превышает значение **SEM_VALUE_MAX**

ENOSYS — если значение аргумента pshared не равно 0, но система не поддерживает *process-shared* семафоры

Этап 3. Атомарные операции

Увеличение счетчика семафора:

int sem_post (sem_t * sem);

Функция sem_post атомарно увеличивает значение счетчика семафора на 1.

Результат:

= 0 — успешное завершение;

-1 — ошибочное завершение, значение счетчика семафора не изменяется, для индикации ошибки используется переменная *errno*

Возможные ошибки:

EINVAL — аргумент sem не является валидным семафором

EOVERFLOW — превышено максимально допустимое значение для счетчика семафора

Уменьшение счетчика семафора

int sem_wait(sem_t * *sem*);

*Функция sem_wait ат*омарно уменьшает значение счетчика семафора (блокирует семафор), на который указывает sem. Если значение семафора больше нуля, то данная операция выполняется и функция немедленно возвращает значение. Если семафор перед вызовом функции имел нулевое значение, то вызов блокируется до тех пор, пока вызов не станет возможным (т. е. пока значение счетчика семафора не поднимается выше нуля) или обработчик сигнала не прерывает вызов.

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

Вариант неблокирующего уменьшения семафора — если операция не может быть немедленно выполнена, то вызов возвращает ошибочное значение (переменная errno принимает значение EAGAIN) вместо блокирования потока.

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

Функция аналогична sem_wait(), за исключением того, что аргумент abs_timeout определяет время блокирования в случае, если операция декремента не может быть немедленно выполнена.

Если указанный момент времени наступил, а семафор не может быть «пройден», то вызов функции sem_timedwait() завершается с ошибкой (переменная errno принимает значение ETIMEDOUT).

Если же операция может быть выполнена немедленно, то функция вообще не обращает внимание на аргумент abs_timeout.

Результат:

= 0 — успешное завершение;

-1 — ошибочное завершение, значение счетчика семафора не изменяется, для индикации ошибки используется переменная *errno*

Возможные ошибки:

EINVAL — аргумент sem не является валидным семафором

EINTR — вызов функции был прерван обработчиком сигнала

Дополнительные ошибки функции sem_trywait():

EAGAIN — операция не может быть выполнена без блокирования (текущее занчение счетчика семафора равно нулю).

Дополнительные ошибки функции sem_timedwait():

EINVAL — значение abs_timeout.tv_nsecs или меньше 0, или больше и равен 1000 million.

ETIMEDOUT — истекло время блокировки, а семафор все еще не может быть захвачен.

Определение текущего значения счетчика семафора sem

```
int sem_getvalue(sem_t *sem, int *sval);
```

Размещает текущее значение счетчика семафора, находящегося по адресу sem, по адресу целочисленной переменной sval.

Если на семафоре заблокированы один или несколько процессов, то в зависимости от системы возможны один из двух вариантов для значения, установленного в sval: либо возвращается 0; или отрицательное число, абсолютное значение которого равно количеству потоков, заблокированных на семафоре. В Linux обычно реализуется принимает первый вариант.

Это не атомарная операция и ее не рекомендуют использовать для управления семафором.

Результат:

= 0 — успешное завершение;

-1 — ошибочное завершение, значение счетчика семафора не изменяется, для индикации ошибки используется переменная *errno*

Возможные ошибки:

EINVAL — аргумент sem не является валидным семафором

Этап 4. Де-инициализация

Когда работа с семафором закончена, его нужно «де-инициализировать»:

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

Семафор sem должен быть «свободен»: если выполнить попытку уничтожить семафор, на котором заблокирован какой-либо поток, то это приведет к непредсказуемому поведению.

Работа с деактивированным семафором приводит к непредсказуемому поведению (семафор должен быть предварительно инициализирован).

Результат:

= 0 — успешное завершение;

-1 — ошибочное завершение, значение счетчика семафора не изменяется, для индикации ошибки используется переменная *errno*

Возможные ошибки:

EINVAL — аргумент sem не является валидным семафором

Такой классический семафор можно использовать как счетчик ресурсов. Если у нас имеется N единиц некоторого ресурса, то для контроля его распределения создается общий семафор S с начальным значением N. Выделение ресурса сопровождается операцией W(S), освобождение — операцией P(S). Значение семафора, таким образом, отражает число свободных единиц ресурса. Если значение семафора 0, то есть, свободных единиц больше не остается, то очередной процесс, запрашивающий единицу ресурса будет переведен в ожидание в операции W(S) до тех пор, пока какой-либо из использующих ресурс процессов не освободит единицу ресурса, выполнив при этом P(S).

Задача производителя-потребителя

```
semaphore lock = 1;                  // для критической секции
semaphore empty_items = size; // 0 — буфер полный, сначала он пустой
semaphore full_items = 0;            // если 0 — буфер пуст
```

```
// производитель
void producer() {
    item = produce();      // создать объект
    W(empty_items);
// есть ли место в буфере?
    W(lock);
// вход в критическую серцию
    append_to_buffer(item);
// добавить объект item в буфер
    P(lock);
// выход из критической секции
    P(full_items);
// уведомить потребителей, что
//  есть новый объект
}
```

```
// потребитель
void consumer() {
    W(full_items);
// не пустой ли буфер?
    W(lock);
// вход в критическую серцию
    item = receive_from_buffer();
// забрать объект item из буфера
    P(lock);
// выход из критической секции
    P(empty_items);
// уведомить производителей, что есть
место
    consume(item);
// «потребить» объект
}
```

Условные переменные

#include <pthread.h>

pthread_cond_t

Этап инициализации

pthread_cond_t cond =
PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond,
const pthread_condattr_t *attr);

Все операции всегда должны выполняться с оригиналом инициализированной условной переменной (Не копии).

Финальный этап де-инициализации

int pthread_cond_destroy(pthread_cond_t *cond);

Де-инициализация условной переменной безопасна только в том случае, если ее не ожидает ни один из потоков.

Условную переменную, де-инициализированную с помощью функции pthread_cond_destroy(), в дальнейшем можно повторно инициализировать с помощью функции pthread_cond_init().

Необходимо де-инициализировать условные переменные, находящиеся в участке динамически выделяемой памяти до освобождения этого участка, а автоматически выделенную условную переменную нужно де-инициализировать до возврата из функции, в которой она определена.

Оповещение и ожидание условных переменных

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex,
                  const struct timespec *abstime);
```

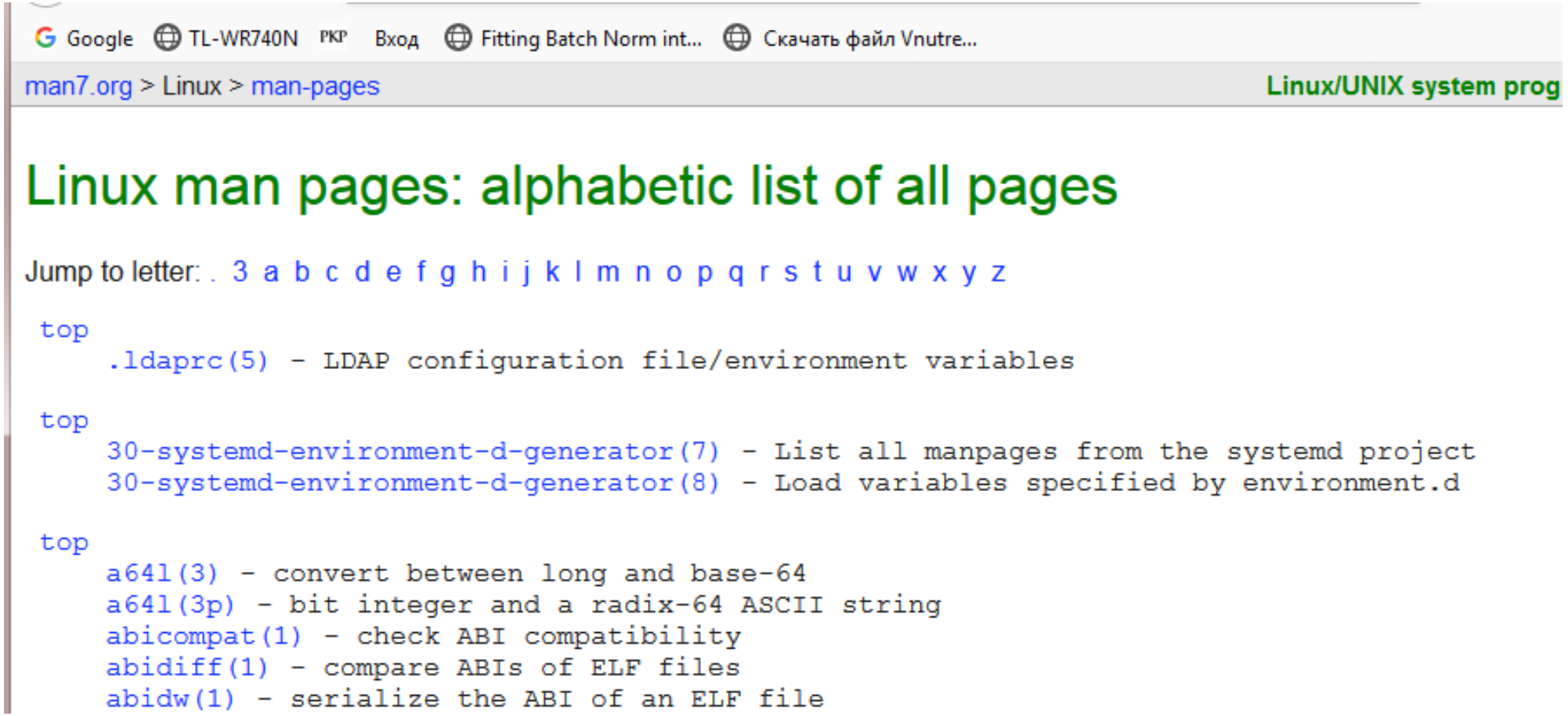
Атрибуты условной переменной

```
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);

int pthread_condattr_getpshared(
        const pthread_condattr_t * attr, int * pshared);
int pthread_condattr_setpshared(
        pthread_condattr_t *attr, int pshared);

pshared :
        PTHREAD_PROCESS_SHARED
        PTHREAD_PROCESS_PRIVATE
```



Более полную информацию можно получить в книгах-справочниках или на специализированых сайтах, например:
http://man7.org/linux/man-pages/dir_all_alphabetic.html