Сергей Юрьевич Шилов

Системное программирование для современных платформ

Системное программирование для современных платформ

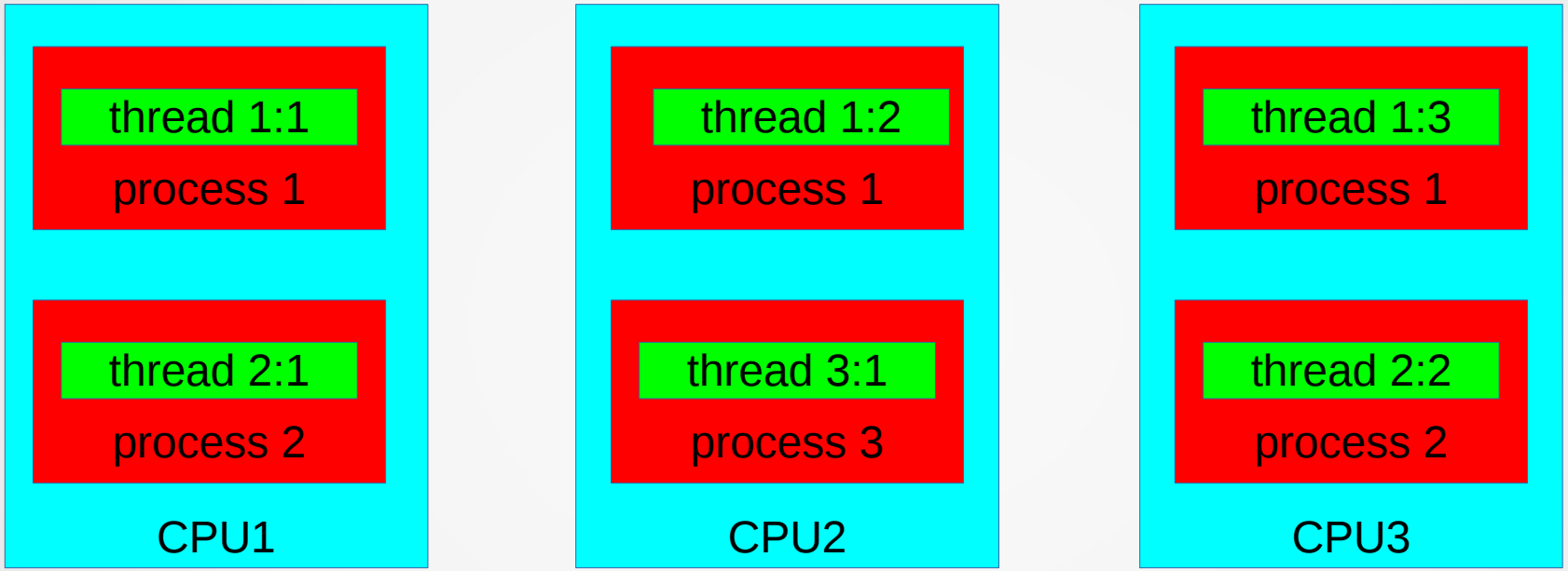
9. Введение в многопоточное программирование

Системное программирование для современных платформ

Исполнение нескольких потоков (нитей) управления в общем адресном пространстве

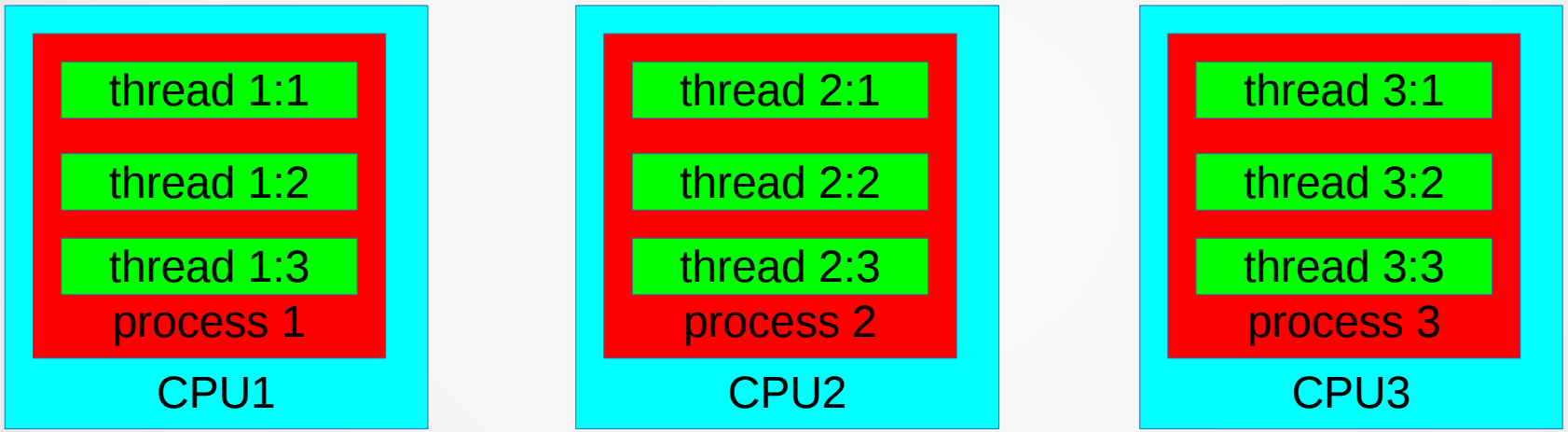
- multistack model
- kernel-, user-, mixed threads
- облегчённые процессы (lightweight processes)
- связанные и несвязанные потоки

Системное программирование для современных платформ



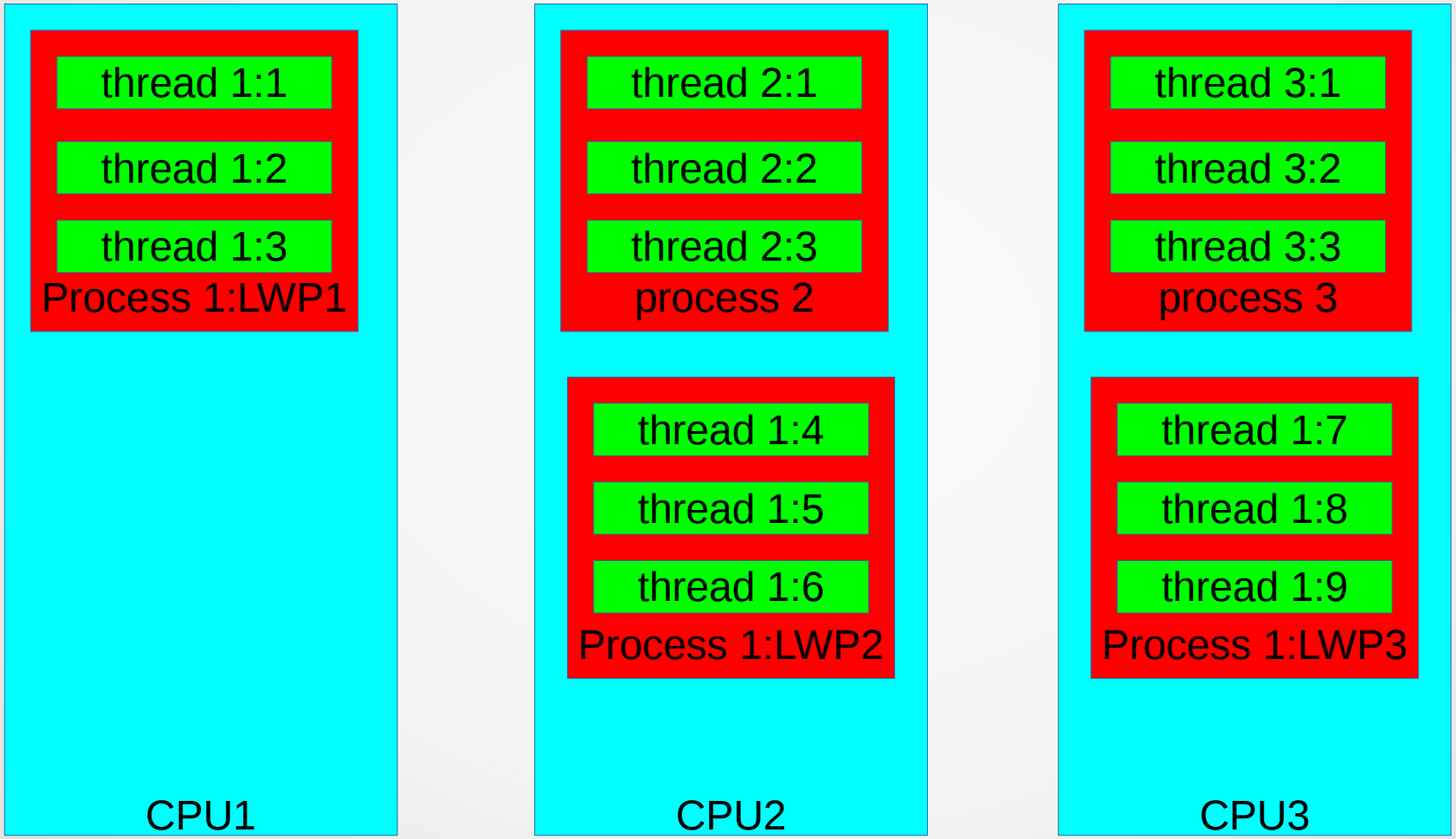
1:1

Системное программирование для современных платформ



N:1

Системное программирование для современных платформ



N:M

Системное программирование для современных платформ

Планировщик ядра	Linux (<=2.6), Windows Win32, FreeBSD (<=6), OpenBSD, NetBSD
Планировщик процесса	MacOSX, DragonFlyBSD
Планировщик библиотеки	Solaris, Linux, FreeBSD, Windows .Net
Гибридный планировщик	Ранние версии NetBSD

Системное программирование для современных платформ

POXIX позволяет работать как планировщиком ядра (PTHREAD_SCOPE_SYSTEM), так и с планировщиком пользовательского уровня (PTHREAD_SCOPE_PROCESS)

Системное программирование для современных платформ

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr, void * (*start_routine)(void *), void
*arg);
```

ОПИСАНИЕ

Функция **pthread_create** служит для создания новой нити исполнения внутри текущего процесса.

Новый **thread** будет выполнять функцию **start_routine** с прототипом

```
void * start_routine(void *);
```

передавая ей в качестве аргумента параметр **arg**. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры. Значение, возвращаемое функцией **start_routine** не должно указывать на динамический объект данного **thread'a**.
Параметр **attr** служит для задания различных атрибутов создаваемого **thread'a**. Их описание выходит за рамки нашего курса и мы вседа будем полагать их заданными по умолчанию, подставляя в качестве аргумента значение **NULL**. Аргумент **attr_p** указывает на структуру, задающую атрибуты вновь создаваемого потока. Если **attr_p=NULL**, то используются атрибуты "по умолчанию" (но это плохая практика, т.к. в разных ОС эти значения могут быть различными, хотя декларируется обратное). Одна структура, указываемая **attr_p**, может использоваться для управления несколькими потоками.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении функция возвращает значение **0** и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр **thread**. В случае ошибки возвращается положительное значение (а не отрицательное, как в большинстве системных вызовов и функций!), которое определяет код ошибки, описанный в файле **errno.h**. Значение системной переменной **errno** при этом не устанавливается.

Системное программирование для современных платформ

```
#include <pthread.h>

int pthread_attr_init (pthread_attr_t *attr_p)
int pthread_attr_destroy (pthread_attr_t *attr_p)
```

ОПИСАНИЕ

Функция **pthread_attr_init** инициализирует структуру, указываемую **attr_p**.

Структура типа **pthread_attr_t** содержит следующие поля (в скобках указаны значения по умолчанию для современных версий **Linux**)

* **scope** - область действия конкуренции [**PTHREAD_SCOPE_PROCESS** - определяет связность потока с **LWP**]. Может принимать значение **PTHREAD_SCOPE_SYSTEM**
* **detachstate** - отсоединенность [**PTHREAD_CREATE_JOINABLE** - определяет то, может или нет какой-либо другой поток ожидать окончания данного (посредством функции)]/ Может принимать значение **PTHREAD_CREATE_DETACHED**
* **stackaddr** - адрес динамического стека потока [**NULL** — адрес определяется операционной системой]
* **stacksize** - размер динамического стека потока [**1 Mb**]
* **schedpolicy** - правила планирования. [**SCHED_OTHER** — политика определяется операционной системой]
* **shedparam** — параметры планирования [**NULL** – по умолчанию]

Функция **pthread_attr_destroy** уничтожает структуру атрибутов потока, на которую указывает аргумент функции.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

При удачном завершении функции возвращают значение **0**. В случае ошибки возвращается другое значение и системной переменной **errno** при этом присваивается соответствующее значение.

Системное программирование для современных платформ

Непосредственно изменять значения полей структуры pthread_attr_t нельзя. Для работы с полями служит специальное API.

	API для проверки значения	API для установки значения
Область конкуренции	pthread_attr_getscope	pthread_attr_setscope
Отсоединённость	pthread_attr_getdetachstate	pthread_attr_setdetachstate
Адрес стека	pthread_attr_getstackaddr	pthread_attr_setstackaddr
Размер стека	pthread_attr_getstacksize	pthread_attr_setstacksize
Правила планировщика	pthread_attr_getshedpolicy	pthread_attr_setshedpolicy
Параметры планировщика	pthread_attr_getshedparam	pthread_attr_setshedparam

Системное программирование для современных платформ

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Описание

Функция **pthread_self** возвращает идентификатор текущей нити исполнения.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Тип данных **pthread_t** является синонимом для одного из целочисленных типов языка C. Функция возвращает идентификатор текущего потока.

Системное программирование для современных платформ

```
#include <pthread.h>

void pthread_exit(void *status);
int pthread_join (pthread_t thread, void **status_addr);
int pthread_detach (pthread_t thread);
```

Описание

Функция **pthread_exit** служит для завершения текущего потока.

Функция никогда не возвращается в вызвавший ее поток. Объект, на который указывает параметр **status**, может быть впоследствии изучен в другом потоке, например, в породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося потока.

Функция **pthread_join** блокирует работу вызвавшего ее потока до завершения потока с идентификатором **thread**. После разблокирования в указатель, расположенный по адресу **status_addr**, заносится адрес, который вернул завершившийся поток либо при выходе из ассоциированной с ним функции, либо при выполнении функции **pthread_exit()**. Если возвращаемое значение не используется, в качестве этого параметра можно использовать значение **NULL**. Поток с идентификатором **thread** не может быть отсоединенным

Функция **pthread_detach** уведомляет планировщик о том, что область памяти для потока **thread** может быть восстановлена, когда он завершит выполнение. Если поток не завершается, функция **pthread_detach** не служит причиной для его завершения. Результат нескольких вызовов функции **pthread_detach** для одного и того же потока не определен.

Необходимость в этой функции возникает по крайней мере в двух случаях:
1. В обработчике запроса на отмену для функции присоединения потока **pthread_join** важно иметь функцию **pthread_detach**, чтобы отсоединить поток. Без нее планировщик вынужден был бы выполнить еще раз функцию **pthread_join**, чтобы попытаться отсоединить поток, который не только задерживает процедуру отмены в течение неограниченного времени, но и вносит новый вызов функции **pthread_join**. В этом случае есть смысл говорить о динамическом отсоединении.
2. Чтобы отсоединить «исходный поток» (это может понадобиться в процессах, которые создают потоки сервера).

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ

Функции **pthread_join** и **pthread_detach** возвращают 0 в случае успеха и другое значение, которое является кодом ошибки в противном случае. Функция **pthread_exit** значения не возвращает.

Системное программирование для современных платформ

Исключающие блокировки (Mutexes

```
#include <pthread.h>
```

int pthread_mutex_init (pthread_mutex_t *mp, const pthread_mutex_attr_t *mattrp) инициализирует взаимоисключающую блокировку, выделяя необходимую память. Если **mattrp=NULL**, то создается блокировка с атрибутами "по умолчанию". В настоящее время атрибут один - область действия блокировки, его умолчательное значение - **PTHREAD_PROCESS_PRIVATE** (тж может быть **PTHREAD_PROCESS_SHARED, поддерживаемое не всеми ОС).**

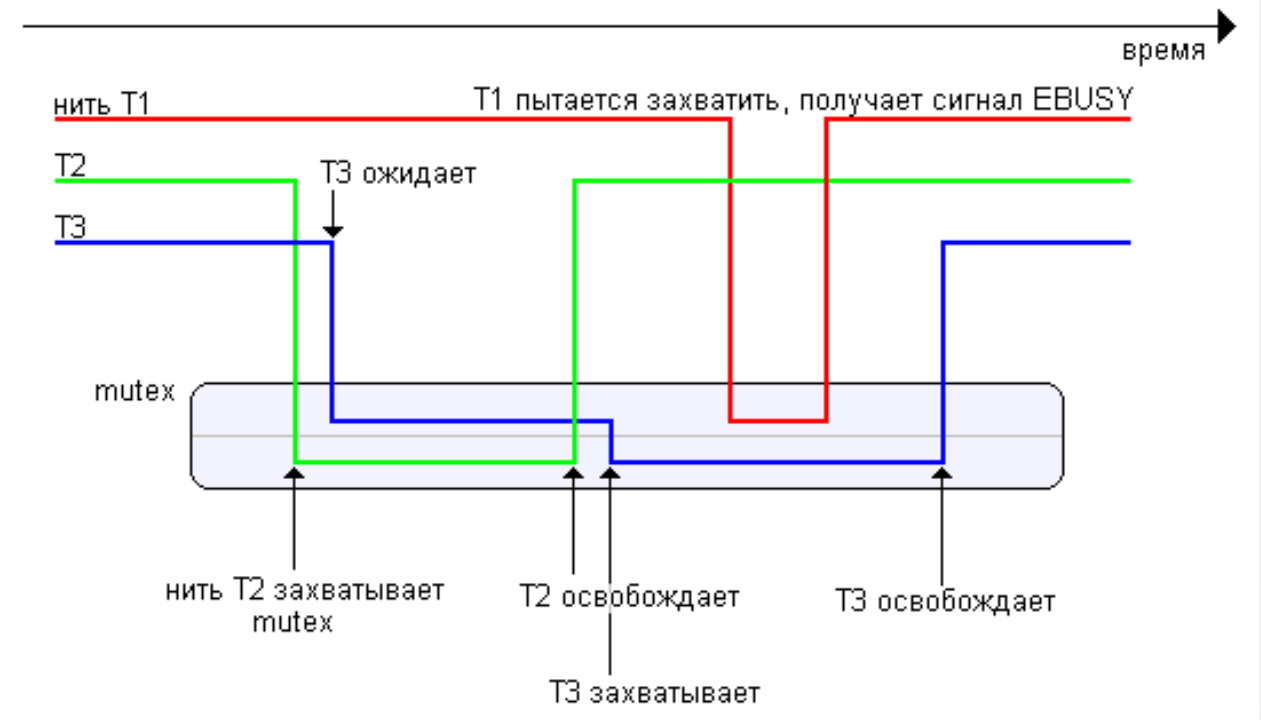
int pthread_mutex_destroy (pthread_mutex_t *mp) - уничтожает блокировку, освобождая выделенную память.

int pthread_mutex_lock (pthread_mutex_t *mp)
int pthread_mutex_unlock (pthread_mutex_t *mp)
int pthread_mutex_trylock (pthread_mutex_t *mp)
С помощью **pthread_mutex_lock** поток пытается захватить блокировку. Если же блокировка уже принадлежит другому потоку, то вызывающий поток ставится в очередь (с учетом приоритетов потоков) к блокировке. После возврата из функции **pthread_mutex_lock** блокировка будет принадлежать вызывающему потоку.

Функция **pthread_mutex_unlock** освобождает захваченную ранее блокировку. Освободить блокировку может только ее владелец.

Функция **pthread_mutex_trylock** - неблокирующая версия функции **pthread_mutex_lock**. Если на момент обращения к этой функции блокировка уже захвачена, то происходит немедленный возврат из функции со значением **EBUSY**.

Системное программирование для современных платформ



Системное программирование для современных платформ

Dead Locks

- Установить иерархию мьютексов и вызывать их строго в порядке от старших к младшим
- Использовать функцию `pthread_mutex_trylock`

Системное программирование для современных платформ

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <errno.h>

static int counter; // shared resource
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void incr_counter(void *p) {
    do {
        usleep(10); // Let's have a time slice between mutex locks
        pthread_mutex_lock(&mutex);
        counter++;
        printf("%d\n", counter);
        sleep(1);
        pthread_mutex_unlock(&mutex);
    } while ( 1 );
}
```

```
void reset_counter(void *p) {
    char buf[10];
    int  num = 0;
    int  rc;
    pthread_mutex_lock(&mutex); // block mutex just to show message
    printf("Enter the number and press 'Enter' to initialize the counter with new value
anytime.\n");
    sleep(3);
    pthread_mutex_unlock(&mutex); // unblock blocked mutex so another thread may work
    do {
        if ( gets(buf) != buf ) return; // NO fool-protection ! Risk of overflow !
        num = atoi(buf);
        if ( (rc = pthread_mutex_trylock(&mutex)) == EBUSY ) {
            printf("Mutex is already locked by another process.\nLet's lock mutex using
pthread_mutex_lock().\n");
            pthread_mutex_lock(&mutex);
        } else if ( rc == 0 ) {
            printf("WOW! You are on time! Congratulation!\n");
        } else {
            printf("Error: %d\n", rc);
            return;
        }
        counter = num;
        printf("New value for counter is %d\n", counter);
        pthread_mutex_unlock(&mutex);
    } while ( 1 );
}

int main(int argc, char ** argv) {
    pthread_t thread_1;
    pthread_t thread_2;
    counter = 0;

    pthread_create(&thread_1, NULL, (void *)&incr_counter, NULL);
    pthread_create(&thread_2, NULL, (void *)&reset_counter, NULL);

    pthread_join(thread_2, NULL);
    return 0;
}
```

Системное программирование для современных платформ

Условные переменные

Применяются в сочетании со взаимоисключающими блокировками. Общая схема использования такова. Один поток устанавливает взаимоисключающую блокировку и затем блокирует себя по условной переменной (путем вызова функции **pthread_cond_wait**), при этом автоматически (но временно) освобождается взаимоисключающая блокировка. Когда какой-либо другой поток посредством вызова функции **pthread_cond_signal** сигнализирует по условной переменной, то первый поток разблокируется и ему возвращается во владение взаимоисключающая блокировка.

int pthread_cond_init (pthread_cond_t *cvp, const pthread_condattr_t *cattrp)
инициализирует условную переменную, выделяя память.

int pthread_cond_destroy (pthread_cond_t *cvp)
уничтожает условную переменную, освобождая память.

int pthread_cond_wait (pthread_cond_t *cvp, const pthread_mutex_t *mp)
автоматически освобождает взаимоисключающую блокировку, указанную mp, а вызывающий поток блокируется по условной переменной, заданной cvp. Заблокированный поток разблокируется функциями **pthread_cond_signal** и **pthread_cond_broadcast**. Одной условной переменной могут быть заблокированы несколько потоков.

int pthread_cond_timedwait (pthread_cond_t *cvp, const pthread_mutex_t *mp, struct timespec *tp)
аналогична функции **pthread_cond_wait**, но имеет третий аргумент, задающий интервал времени, после которого поток разблокируется (если этого не было сделано ранее).

int pthread_cond_signal (pthread_cond_t *cvp)
разблокирует ожидающий данную условную переменную поток. Если сигнала по условной переменной ожидают несколько потоков, то будет разблокирован только какой-либо один из них.

int pthread_cond_broadcast (pthread_cond_t *cvp)
разблокирует все потоки, ожидающие данную условную переменную.

Системное программирование для современных платформ

Барьеры

Барьер используется для синхронизации работы нескольких потоков управления. Барьер характеризуется натуральным числом **count**, задающим количество синхронизируемых потоков. Поток управления, "подошедший" к барьеру (обратившийся к функции **pthread_barrier**), блокируется до момента накопления перед этим барьером указанного количества потоков **count**.

int pthread_barrier_init(pthread_barrier_t *bp, pthread_barrierattr_t *attr, unsigned count)
инициализирует барьер, выделяя необходимую память, устанавливая значения его атрибутов и назначая **count** "шириной" барьера. В настоящее время атрибуты барьеров не определены поэтому в качестве второго параметра функции **pthread_barrier_init** следует использовать NULL.

int pthread_barrier_destroy(pthread_barrier_t *bp)
уничтожает барьер, освобождая выделенную память.

int pthread_barrier_wait(pthread_barrier_t *bp)
приостанавливает вызвавший данную функцию поток до момента накопления перед барьером **count** потоков. Заблокированный поток может быть прерван сигналом, при этом обработчик сигнала (если он был назначен) будет вызван на выполнение обычным образом. Выход из обработчика вернет поток в состояние ожидания, если к этому моменту требуемое количество **count** потоков еще не скопилось перед барьером.

Системное программирование для современных платформ

КОНЕЦ

Спасибо за внимание ;)