14.3. Многопотоковое программирование

Многопотоковое программирование предложено в качестве средства разработки параллельных программ для многопроцессорных систем (систем с разделяемой памятью). При этом реальное разнесение потоков управления на разные процессоры - задача ОС. Фирма SUN Microsystems для поддержки потоков (нитей) управления реализовала *легковесные процессы* LWP (LightWeight Processes). Диспетчирование LWP - практически не управляемая пользователем процедура. Потоки характеризуются следующими атрибутами:

- идентификатор потока (уникален в рамках процесса);
- значение приоритета;
- сигнальная маска.

Предложены 2 API потокового программирования:

- фирмы SUN Microsystems (пионер в этом деле);
- комитета POSIX.1C по стандартизации.

Здесь рассматривается вариант POSIX. Все функции этого варианта имеют в своих именах префикс pthread_ и объявлены в заголовочном файле pthread.h.

14.3.1. Создание потока управления

```
int pthread_create (pthread_t *tid_p, const pthread_attr_t *attr_p,
                    void *(*func_p)(void *), void *arg_p)
```

Создает новый поток для функции, заданной параметром *func_p*. Эта функция имеет аргументом указатель (void *) и возвращает значение того же типа. Реально же в функцию передается аргумент *arg_p*. Идентификатор нового потока возвращается через *tid_p*.

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.

14.3.2. Инициализация атрибутов потока

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

Инициализирует структуру, указываемую *attr_p*, значениями "по умолчанию" (при этом распределяется кое-какая память).

Не будем обсуждать все атрибуты и подробности их использования, дадим лишь список и поясним два из них.

1. Область действия конкуренции (scope) [PTHREAD_SCOPE_PROCESS] - определяет связность потока с LWP.
2. Отсоединенность (detachstate) [PTHREAD_CREATE_JOINABLE] - определяет то, может или нет какой-либо другой поток ожидать окончания данного (посредством функции).
3. Адрес динамического стека потока (stackaddr) [NULL].
4. Размер динамического стека потока(stacksize) [1 Mb].
5. Приоритет потока (priority) [наследуется от потока-родителя].
6. Правила и параметры планирования. Неприятно то, что schedpolicy по умолчанию устанавливается в SCHED_OTHER, зависимую от ОС.

14.3.3. Освобождение памяти атрибутов потока

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

14.3.4. Область конкуренции

```
int pthread_attr_setscope (pthread_attr_t *attr_p, int scope)
int pthread_attr_getscope (pthread_attr_t *attr_p, int *scope)
```

scope может принимать два значения:
PTHREAD_SCOPE_PROCESS - для *несвязанного* потока;
PTHREAD_SCOPE_SYSTEM - для *связанного* потока.

14.3.5. Состояние отсоединенности

```
int pthread_attr_setdetachstate (pthread_attr_t *attr_p, int detachstate)
int pthread_attr_getdetachstate (pthread_attr_t *attr_p, int *detachstate)
```

detachstate может принимать два значения:
PTHREAD_CREATE_DETACHED - для *отсоединеного* потока;
PTHREAD_CREATE_JOINABLE - для *присоединенного* потока.

Для отсоединенного потока невозможно его ожидание его окончания другим потоком, поэтому после окончания такого потока все его ресурсы могут быть освобождены (и использованы заново).

14.3.6. Завершение потока

В потоках можно использовать стандартную функцию exit(), однако это ведет к немедленному завершению всех потоков и процесса в целом.

Поток завершается вместе с вызовом return() в функции, вызванной pthread_create().

Поток заканчивает свое выполнение также с помощью функции

```
pthread_exit (void *status),
```

допустимо в качестве *status* использовать NULL.

Поток может быть завершен другим потоком посредством функции pthread_cancel() (с этой функцией работают pthread_setcanceltype, pthread_setcancelstate и pthread_testcancel).

14.3.7. Ожидание завершения потока

```
int pthread_join (pthread_t tid, void **status)
```

Вызывающий поток блокируется до окончания потока с идентификатором *tid*. Поток с идентификатором *tid* не может быть отсоединенным

14.3.8. Получение идентификатора потока

```
pthread_t pthread_self (void)
```

14.3.9. Передача управления другому потоку

`int sched_yield (void)`

Передает управление другому потоку, имеющему приоритет равный или больший приоритета вызывающего потока.

14.3.10. Посылка сигнала потоку

`int pthread_kill (pthread_t tid, int signum)`

Посылает сигнал с идентификатором *signum* в поток, задаваемый идентификатором *tid*.

14.3.11. Манипулирование сигнальной маской потока

`int pthread_sigmask (int mode, sigset_t *set_p, sigset_t *old_p)`

Изменяет сигнальную маску потока в соответствии с аргументом *mode*, который может принимать следующие значения:

- SIG_BLOCK - добавить сигналы из набора, указываемого *set_p*, в текущую сигнальную маску, описывающую блокируемые сигналы;
- SIG_UNBLOCK - удалить сигналы, содержащиеся в наборе, указываемом *set_p*, из текущей сигнальной маски;
- SIG_SETMASK - установить сигнальную маску, указываемую *set_p*, в качестве текущей.

Если значение аргумента *old_p* не равно NULL, то в область памяти, указываемую *old_p*, помещается предыдущее содержимое сигнальной маски.

14.3.12. Объекты синхронизации потоков управления

Потоки используют единое адресное пространство. Это означает, что все *статические* переменные доступны потокам в любой момент. Поэтому необходимы средства управления доступом к совместно используемым данным. Здесь возможно использование *стандартных* средств синхронизации различных *процессов*: каналы, очереди сообщений, межпроцессные семафоры. Однако, специально для межпотокового взаимодействия предложены индивидуальные средства:

- взамоисключающие блокировки (mutex locks);
- условные переменные (conditional variables);
- семафоры (semaphores).

Указанные средства перечислены в порядке ухудшения их эффективности.

Заметим, что доступ к атомарным данным (char, int, double) реализуется за один такт процессора, поэтому существуют ситуации (зависящие от логики программы), когда такие данные сами могут выступать в качестве средства синхронизации.

14.3.13. Взамоисключающие блокировки

`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 (а может быть еще PTHREAD_PROCESS_SHARED).

int pthread_mutex_destroy (pthread_mutex_t *mp)

разрушает 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.

14.3.14. 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)

разрушает 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.

14.3.15. Семафоры

Семафор представляет собой целочисленную переменную. Потоки могут наращивать (post) и уменьшать (wait) ее значение на единицу. Если поток пытается уменьшить семафор так, что его значение становится отрицательным, то поток блокируется. Поток будет разблокирован, когда какой-либо другой поток не увеличит значение семафора так, что он станет *неотрицательным* после уменьшения его первым (заблокированным) потоком.

Потоки похожи на взаимоисключающие блокировки и условные переменные, но отличаются от них тем, что у них нет "владельца", т.е. изменить значение семафора может *любой* поток.

В POSIX-версии средств многопотокового программирования используются те же самые семафоры, что и для межпроцессного взаимодействия.

#include <semaphore.h>
int sem_init (sem_t *_sp_, int _pshared_, unsigned int _value_)

инициализирует семафор, указанный аргументом *sp*, значением *value*. Если *pshared*=0, то область действия семафора - только один процесс, иначе - несколько процессов.

int sem_destroy (sem_t *_sp_)

разрушает семафор.

int sem_post (sem_t *_sp_)

увеличивает значение семафора на 1, при этом может быть разблокирован один (из, возможно, нескольких) поток (какой именно не определено).

int sem_wait (sem_t *_sp_)

пытается уменьшить значение семафора на 1. Если при этом значение семафора должно стать отрицательным, то поток блокируется.

int sem_trywait (sem_t *_sp_)

неблокирующая версия функции sem_wait().

Литература:

D. P. Bovet, M. Cesati, Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005
W. R. Stevens, S. A. Rago, Advanced Programming in the UNIX® Environment: Second Edition, Addison Wesley Professional, 2005