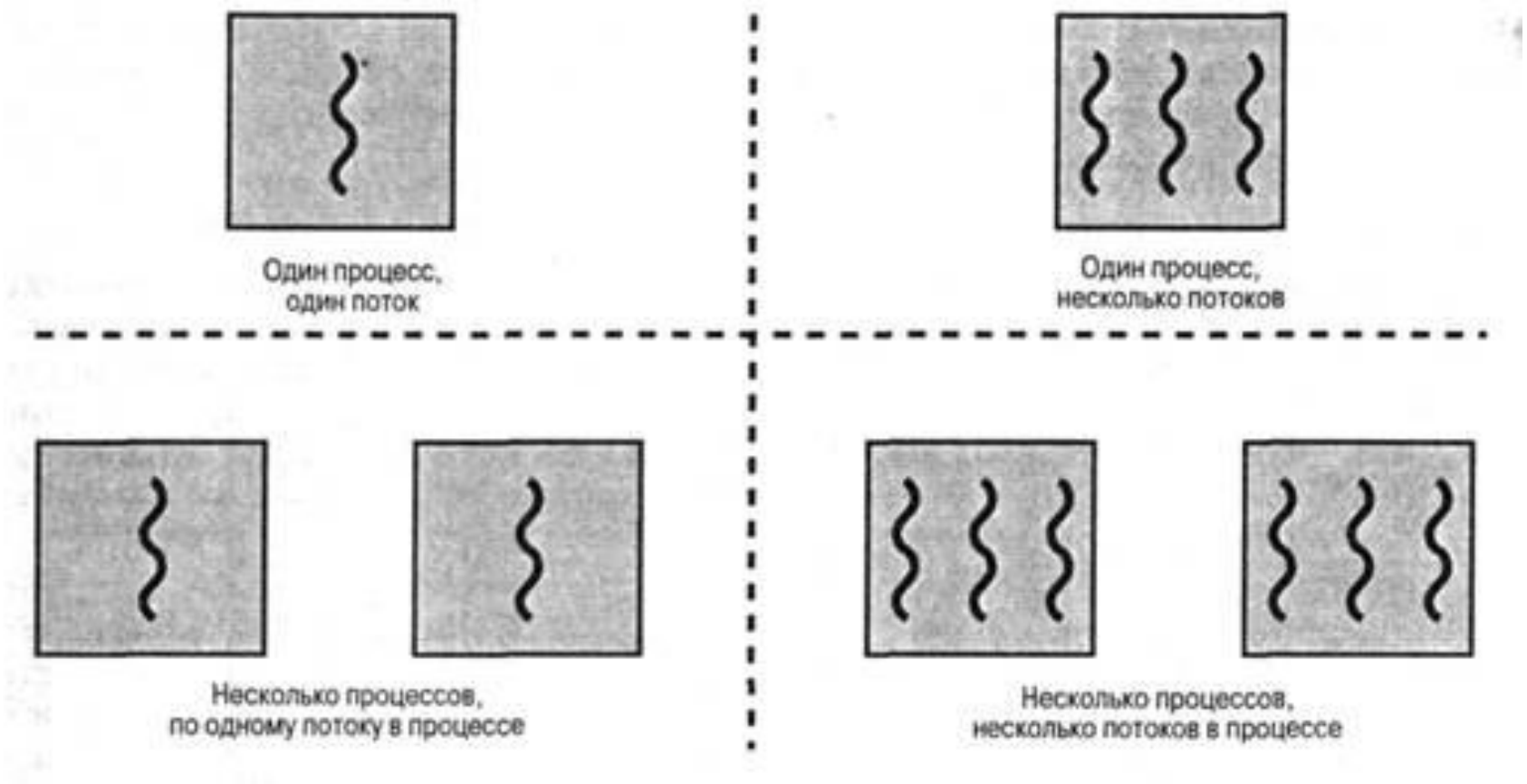
Архитектура ЭВМ. Операционные системы

Власов Евгений

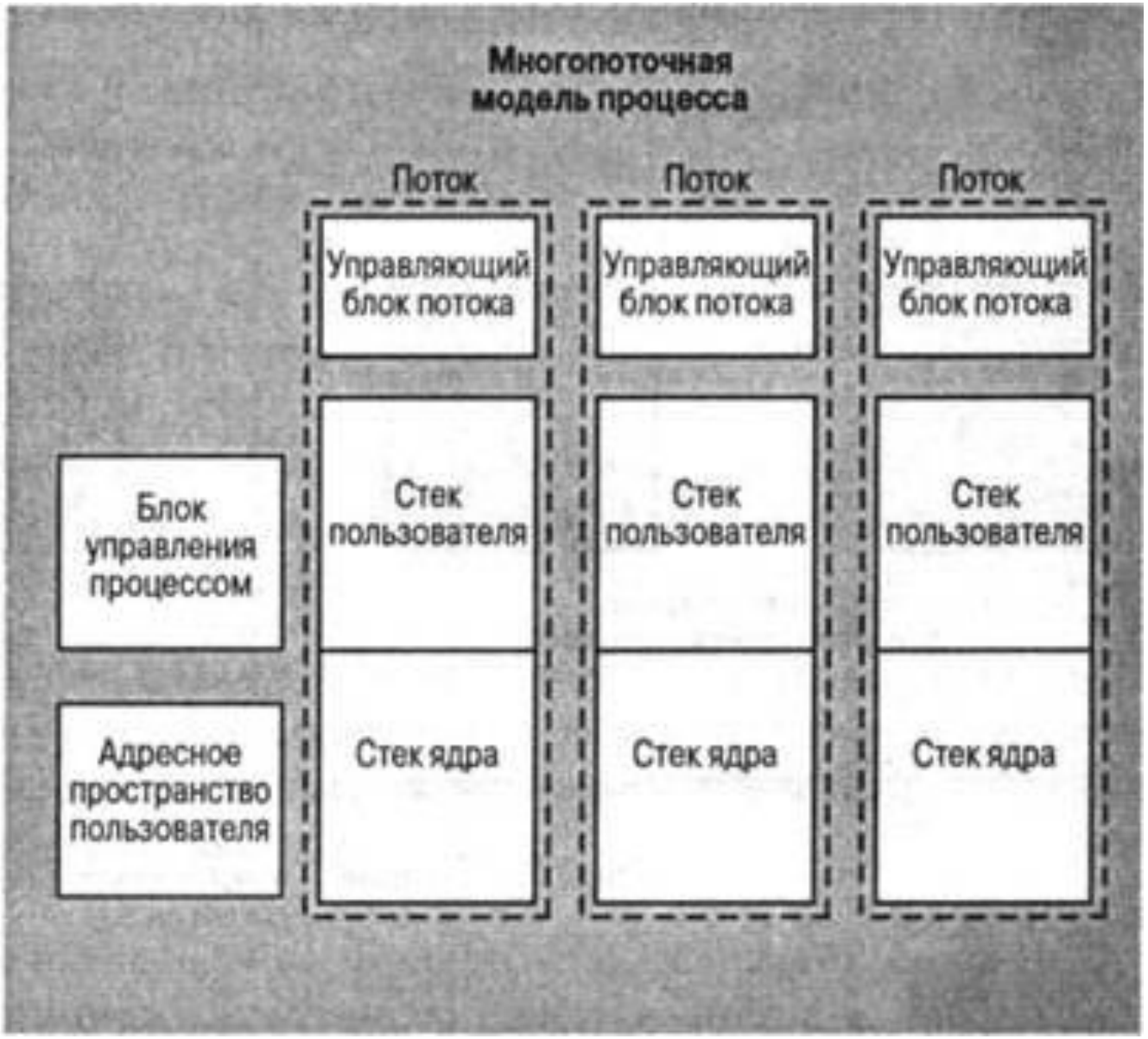
Потоки исполнения - Thread

Наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Реализация потоков выполнения и процессов в разных операционных системах отличается друг от друга, но в большинстве случаев поток выполнения исполняется в рамках процесса. Несколько потоков выполнения могут существовать в рамках одного и того же процесса и совместно использовать ресурсы, такие как память, тогда как процессы не разделяют этих ресурсов. В частности, потоки выполнения разделяют инструкции процесса (его код) и его контекст (значения переменных, которые они имеют в любой мо-мент времени).

Для понимания работы потоков необходимо пом- нить, что каждый поток имеет независимый от других потоков стек выполне- ния.




Однопоточная
модель процесса
Блок управления процессом
Адресное пространство пользователя
Стек пользователя
Стек ядра


Многопоточная
модель процесса
Поток
Поток
Поток
Управляющий блок потока
Управляющий блок потока
Управляющий блок потока
Блок управления процессом
Адресное пространство пользователя
Стек пользователя
Стек пользователя
Стек пользователя
Стек ядра
Стек ядра
Стек ядра

В рамках процесса могут находиться один или несколько потоков, каждый из которых обладает следующими характеристиками:

- Состояние выполнения потока (выполняющийся, готовый к выполнению и т.д.).
- Сохраненный контекст не выполняющегося потока; один из способов рассмотрения потока — считать его независимым счетчиком команд, работающим в рамках процесса.
- Стек выполнения
- Статическая память, выделяемая потоку для локальных переменных.
- Доступ к памяти и ресурсам процесса, которому этот поток принадлежит; этот доступ разделяется всеми потоками данного процесса.

Потоки выполнения отличаются от традиционных процессов многозадачной операционной системы тем, что:

- процессы, как правило, независимы, тогда как потоки выполнения су-ществуют как составные элементы процессов
- процессы несут значительно больше информации о состоянии, тогда как несколько потоков выполнения внутри процесса совместно используют информацию о состоянии, а также память и другие вычислительные ресурсы
- процессы имеют отдельные адресные пространства, тогда как потоки выполнения совместно используют их адресное пространство
- процессы взаимодействуют только через предоставляемые системой механизмы связей между процессами
- переключение контекста между потоками выполнения в одном процессе, как правило, быстрее, чем переключение контекста между процессами.

Основные преимущества использования потоков с точки зрения производительности:

- Создание нового потока в уже существующем процессе занимает намного меньше времени, чем создание совершенно нового процесса. Скорость создания процессов по сравнению с такой же скоростью в UNIX-совместимых приложениях, в которых не используются потоки, возрастает в ~10 раз.
- Поток можно завершить намного быстрее, чем процесс.
- Переключение потоков в рамках одного и того же процесса происходит намного быстрее.
- При использовании потоков повышается эффективность обмена информацией между двумя выполняющимися программами. В большинстве операционных систем обмен между независимыми процессами происходит с участием ядра, в функции которого входит обеспечение защиты и механизма, необходимого для осуществления обмена. Однако благодаря тому, что различные потоки одного и того же процесса используют одну и ту же область памяти и одни и те же файлы, они могут обмениваться

Примеры многопоточных программ

- Фоновая обработка данных в программах с развитым интерфейсом пользователя
- Асинхронная обработка данных от нескольких источников
- Текстовые редакторы для автоматического сохранения копий документа

Системные вызовы для работы потоков

Название	**Выполняемые действия**
`pthread_create`	Создает новый поток
`pthread_self`	Возвращает идентификатор текущего потока
`pthread_exit`	Завершает поток
`pthread_kill`	Посылает сигнал в поток
`pthread_join`	Блокирует вызывающий поток до за- вершения потока

Системный вызов pthread_create

```
#include <pthread.h>
int pthread_create(pthread_t *tid,
          const pthread_attr_t *attr,
void*(*func) (void*),
          void *arg);
```

При успешном завершении идентификатор созданного потока помещается по адресу, указанному параметром `tid`. Атрибуты создаваемого потока передаются в attr. Если attr равен NULL, то используются параметры по умолчанию.

Параметр `func` передает указатель на функцию следующего вида `void thread_func(void*)` Аргумент `arg` передается в функцию func.

Системный вызов `pthread_self`

```
#include <pthread.h>
pthread_t pthread_self();
```

Возвращает идентификатор текущего потока

Системный вызов `pthread_join`

```
#include <pthread.h>
int pthread_join (pthread_t thread, void
**status_addr);
```

Функция `pthread_join` блокирует работу вызвавшей ее нити исполне- ния до завершения thread'а с идентификатором thread. После разбло- кирования в указатель, расположенный по адресу `status_addr`, заносится адрес, который вернул завершившийся thread либо при выходе из ассоциированной с ним функции, либо при выполнении функции `pthread_exit()`. Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение NULL.

Системный вызов `pthread_detach`

```
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

Отсоединяет поток. По умолчанию все потоки создаются присоединенными. Это означает, что когда поток завершается его идентификатор и статус завершения сохраняются до тех пор, пока какой-либо поток данного процесса не вызовет pthread_join. Если поток является отсоединенным, то после его за- вершения все ресурсы освобождаются.

Системный вызов `pthread_exit`

```
#include <pthread.h>
void pthread_exit(void *status);
```

Функция `pthread_exit` служит для завершения нити исполнения текущего процесса.

Функция никогда не возвращается в вызвавший ее thread. Объект, на который указывает параметр status, может быть впоследствии изучен в другой нити исполнения, например, в породившей завершившуюся нить. Поэтому он не должен указывать на динамический объект завершившегося thread'а.

Системный вызов `pthread_cancel`

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

Завершает досрочно поток tid. Может завершить поток досрочно, ее нельзя назвать средством принудительного завершения потоков. Поток может не только самостоятельно выбрать порядок завершения в ответ на вызов `pthread_cancel()`, но и вовсе игнорировать этот вызов. Вызов функции `pthread_cancel()` следует рассматривать как запрос на выполнение досрочного завершения потока.

Системный вызов `pthread_setcancelstate()`

```
#include <pthread.h>
Int pthread_setcancelstate(int state, int
*oldstate);
```

Устанавливает режим реагирования потока на вызов `pthread_cancel()`.

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
//Здесь поток завершать нельзя
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
```

Значение режима реагирования на `pthread_cancel`

PTHREAD_CANCEL_ENABLE	Поток может быть прерван (по умолчанию)
PTHREAD_CANCEL_DISABLE	Поток не может быть прерван. Если запрос на прерывание получен, то он будет игнорироваться до тех пор пока не будет возвращение состояние **PTHREAD_CANCEL_ENABLE**

Системный вызов pthread_setcanceltype()

#include <pthread.h>
Int pthread_setcanceltype(int type, int *oldtype);

Устанавливает *тип* реагирования на прерывание.

PTHREAD_CANCEL_DEFERRED	Заставляющее запросы на отмену ждать, пока поток не выполнит одну из функций, проверящюих необходимость прерывания.
PTHREAD_CANCEL_ASYNCHRONOUS	Завершение выполняется немедленно

Системный вызов `pthead_testcancel()`

```
#include <pthread.h>
Void pthread_testcancel();
```

Создает точку проверки прерывания потока.

Точки прерывания

Функции, которые проверяют нужно ли прерывать поток:
accept(), aio_suspend(), close(), connect(), creat(), fcntl(), fsync(), lockf(), msgrcv(), msgsnd(), msync(), nanosleep(), open(), pause(), poll(), pread(), pselect(), pthread_cond_timedwait(), pthread_cond_wait(), pthread_join(), pthread_testcancel(), pwrite(), read(), readv(), recv(), recvfrom(), recvmsg(), select(), sem_wait(), send(), sendmsg(), sendto(), sigpause(), sigsuspend(), sigwait(), sleep(), system(), tcdrain(), usleep(), wait(), waitpid(), write(), writev().

Механизм отложенного досрочного завершения очень полезен, но для действительно эффективного управления завершением потоков необходим еще и механизм, оповещающий поток о досрочном завершении. Оповещение о завершении потоков в Unix-системах реализовано на основе тех же принципов, что и оповещение о завершении самостоятельных процессов. Если нам нужно выполнять какие-то специальные действия в момент завершения потока (нормального или досрочного), мы устанавливаем функцию-обработчик, которая будет вызвана перед тем, как поток завершит свою работу. Для потоков наличие обработчика завершения даже более важно, чем для процессов. Предположим, что поток выделяет блок динамической памяти и затем внезапно завершается по требованию другого потока. Если бы поток был самостоятельным процессом, ничего особенно неприятного не случилось бы, так как система сама убрала бы за ним мусор. В случае же процесса-потока не высвобожденный блок памяти так и останется «висеть» в адресном пространстве многопоточного приложения. Если потоков много, а ситуации, требующие досрочного завершения, возникают часто, утечки памяти могут оказаться значительными. Устанавливая обработчик завершения потока, высвобождающий занятую память, мы можем быть уверены, что поток не оставит за собой бесхозных блоков памяти (если, конечно, в системе не случится какого-то более серьезного сбоя).

Для установки обработчика завершения потока применяется макрос `pthread_cleanup_push()`. Подчеркиваю жирной красной чертой, `pthread_cleanup_push()` – это **макрос, а не функция**. Неправильное использование макроса `pthread_cleanup_push()` может привести к неожиданным синтаксическим ошибкам. У макроса pthread_cleanup_push() два аргумента. В первом аргументе макросу должен быть передан адрес функции-обработчика завершения потока, а во втором – нетипизированный указатель, который будет передан как аргумент при вызове функции-обработчика. Этот указатель может указывать на что угодно, мы сами решаем, какие данные должны быть переданы обработчику завершения потока. Макрос `pthread_cleanup_push()` помещает переданные ему адрес функции- обработчика и указатель в специальный стек. Само слово «стек» указывает, что мы можем назначить потоку произвольное число функций-обработчиков завершения. Поскольку в стек записывается не только адрес функции, но и ее аргумент, мы можем назначить один и тот же обработчик с несколькими разными аргументами.

В процессе завершения потока функции-обработчики и их аргументы должны быть извлечены из стека и выполнены. Извлечение обработчиков из стека и их выполнение может производиться либо явно, либо автоматически. Автоматически обработчики завершения потока выполняются при вызове потоком функции `pthread_exit()`, завершающей работу потока, а также при выполнении потоком запроса на досрочное завершение. Явным образом обработчики завершения потока извлекаются из стека с помощью макроса `pthread_cleanup_pop`. Во всех случаях обработчики извлекаются из стека (и выполняются) в порядке, противоположном тому, в котором они были помещены в стек. Если мы используем макрос `pthread_cleanup_pop()` явно, мы можем указать, что обработчик необходимо только извлечь из стека, но выполнять его не следует.