

Лекция 4. Потоки

CS221. Архитектура компьютера и операционные системы

12 октября 2017 г.

Применение многопоточности

Основные области применения

- Параллельные алгоритмы.
- Сетевые приложения, асинхронный ввод/вывод.
- Пользовательский интерфейс.

Основной цикл обработки сообщений

Пример

```
#include <windows.h>

int APIENTRY WinMain(
    HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpszCmdLine, int nCmdShow)
{
    MSG cMsg;
    // Создание основного окна и т. д.
    while (GetMessage(&cMsg, NULL, 0, 0))
        DispatchMessage(&cMsg);
    //
    return cMsg.wParam;
}
```

Планирование

Определение

Политика планирования: (Scheduling Strategy) — набор правил, определяющих, когда и как выбирается новый процесс для выполнения на соответствующем процессорном ядре.

Виды планирования

- краткосрочное;
- долгосрочное.

Критерии планирования задач

Пользовательские критерии (производительность)

Время оборота: (*turnaround time, TAT*) интервал между запуском и завершением процесса.

Время отклика: (*response time, RT*) интервал между подачей запроса и началом выдачи ответа.

Предельный срок: (*deadline, DL*) момент, до которого процесс должен быть завершён.

Пользовательские критерии (прочее)

Предсказуемость: выполнение задания примерно за одинаковое время при разных запусках.

Критерии планирования задач

Пользовательские критерии (производительность)

Время оборота: (*turnaround time, TAT*) интервал между запуском и завершением процесса.

Время отклика: (*response time, RT*) интервал между подачей запроса и началом выдачи ответа.

Предельный срок: (*deadline, DL*) момент, до которого процесс должен быть завершён.

Пользовательские критерии (прочее)

Предсказуемость: выполнение задания примерно за одинаковое время при разных запусках.

Критерии планирования задач (окончание)

Системные критерии (производительность)

Пропускная способность: (*throughput, TP*) количество процессов, завершающихся за единицу времени.

Использование процессорных ресурсов: средняя доля времени, в течение которого занято процессорное ядро.

Системные критерии (прочее)

Беспристрастность: (*fairness*) все процессы должны рассматриваться как равноправные при отсутствии дополнительных указаний.

Использование приоритетов: предпочтение в обслуживании процессам с более высоким приоритетом.

Баланс ресурсов: предпочтение в обслуживании процессам с малым использованием системных ресурсов.

Критерии планирования задач (окончание)

Системные критерии (производительность)

Пропускная способность: (*throughput, TP*) количество процессов, завершающихся за единицу времени.

Использование процессорных ресурсов: средняя доля времени, в течение которого занято процессорное ядро.

Системные критерии (прочее)

Беспристрастность: (*fairness*) все процессы должны рассматриваться как равноправные при отсутствии дополнительных указаний.

Использование приоритетов: предпочтение в обслуживании процессам с более высоким приоритетом.

Баланс ресурсов: предпочтение в обслуживании процессам с малым использованием системных ресурсов.

Проблемы планирования

Определение

Голодная смерть: (*starvation*) — ситуация невозможности завершения процесса из-за постоянного отказа ему в требуемом ресурсе (процессорное время, ...) со стороны операционной системы.

Невытесняющие алгоритмы

Определение

«Первым поступил, первым обслужен»: (*First Come, First Served, FCFS*) — процессы выполняются без вытеснения в порядке очереди: $\max w$.

«Сначала кратчайшая задача»: (*Shortest Job First, SJF, SPN*) — аналогично, очерёдность от самого короткого процесса: $\min s$.

«Наивысшее время отклика»: (*Highest Ratio Response Next, HRRN*) — аналогично, очерёдность от процесса с наивысшим отношением отклика: $\max((s + w)/s)$.

Вытесняющие алгоритмы

Определение

Карусельное планирование: (*Round-Robin scheduling, RR*) — процессы выполняются в течение фиксированного кванта по кругу с вытеснением.

Вытесняющее планирование с фиксированным приоритетом: (*Fixed priority pre-emptive scheduling*) — аналогично, выбирается готовый процесс с наивысшим приоритетом.

Многоуровневая очередь с обратной связью: (*multilevel FeedBack queue, FB*):

- 1 Новый процесс попадает в очередь с наивысшим приоритетом.
- 2 Добровольно освободивший квант времени процесс попадает в ту же очередь.
- 3 Полностью исчерпавший свой квант процесс попадает в конец очереди со следующим приоритетом.

Пример планирования процессов

Пример

Процесс	Время запуска	Время обслуживания
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Диаграммы Ганта (Gantt diagrams)

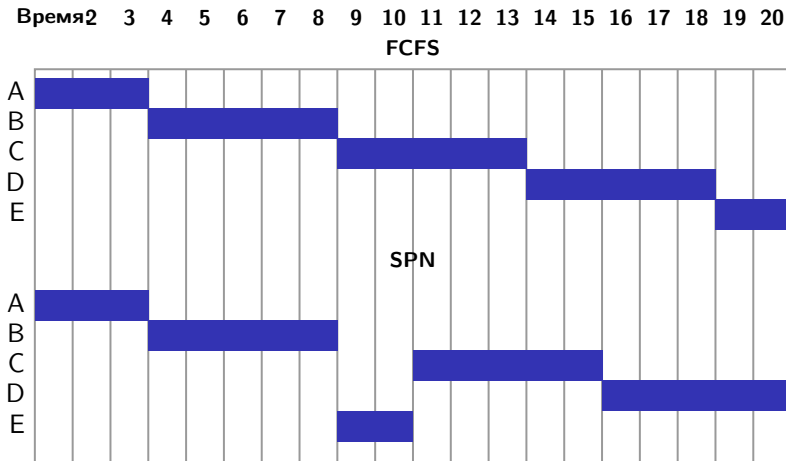


Рис. 1: диаграммы Ганта выполнения процессов при различных стратегиях

Диаграммы Ганта (окончание)

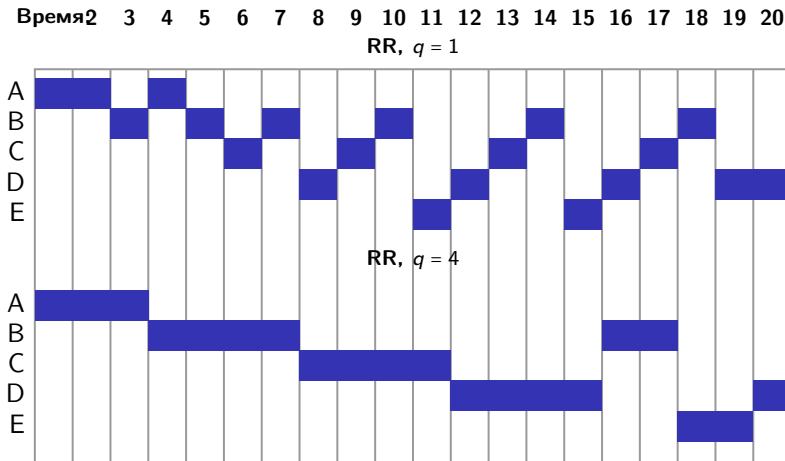


Рис. 2: диаграммы Ганта выполнения процессов при различных стратегиях

Сравнение алгоритмов планирования

Алгоритм	CPU	TP	TAT	RT	NoS	DL
First come, first served (FCFS)	↓	↓	↑	↑	+	-
Shortest Job First (SJF)	↕	↑	↕	↕	-	-
Highest ratio response next (HRRN)	↑	↑	↑	↑	-	-
Round-robin scheduling (RR)	↑	↕	↕	↑	+	-
Fixed priority pre-emptive sched.	↕	↓	↕	↕	-	+
Multilevel feedback queue (FB)	↑	↑	↕	↕	+	+

Таблица 1: характеристики классических алгоритмов планирования

Приоритеты

```
IDLE_PRIORITY_CLASS  
BELOW_NORMAL_PRIORITY_CLASS  
NORMAL_PRIORITY_CLASS  
ABOVE_NORMAL_PRIORITY_CLASS  
HIGH_PRIORITY_CLASS  
REALTIME_PRIORITY_CLASS
```

Таблица 2: классы приоритета процессов

```
THREAD_PRIORITY_IDLE  
THREAD_PRIORITY_LOWEST  
THREAD_PRIORITY_BELOW_NORMAL  
THREAD_PRIORITY_NORMAL  
THREAD_PRIORITY_ABOVE_NORMAL  
THREAD_PRIORITY_HIGHEST  
THREAD_PRIORITY_TIME_CRITICAL
```

Таблица 3: уровни приоритета потоков

Базовый приоритет потока

Определение

Диспетчер ядра: (Kernel Dispatcher) — набор процедур ядра, выполняющих обязанности планировщика задач.

№	№	Описание
0		поток, обнуляющий страницы
1	– 15	варьируемые (динамические) уровни
16	– 31	уровни реального времени

Таблица 4: уровни приоритета потоков

Вычисление приоритетов

Класс/Уровень	Idle	Below normal	Normal	Above normal	High	Realtime
Idle: -16	1	1	1	1	1	16
Lowest: -2	2	4	6	8	11	22
Below normal: -1	3	5	7	9	12	23
Normal: 0	4	6	8	10	13	24
Above normal: +1	5	7	9	11	14	25
Highest: +2	6	8	10	12	15	26
Time critical: +16	15	15	15	15	15	31

Таблица 5: отображение классов приоритетов процессов/уровней приоритетов потоков на приоритеты

Состояния потоков

Состояния потоков в планировщике

Инициализирован: (*initialized*) — в начале после создания.

Готов: (*ready*) — ожидает выполнения.

Простаивает: (*standby*) — выбран следующим для исполнения на конкретном процессоре.

Готов, отложен: (*deferred ready*) — (Windows Server 2003 и выше) выбран для выполнения на процессоре, но ещё не запланирован.

Выполняется: (*running*) — выполняется, пока на него переключён контекст.

Ожидает: (*waiting*) — самостоятельно начинает ожидание или его вынуждает к этому подсистема окружения.

Переходное состояние: (*transition*) — готов к выполнению, но его стек ядра выгружен из памяти.

Завершён: (*terminated*) — после завершения выполнения.

Состояния потоков (окончание)

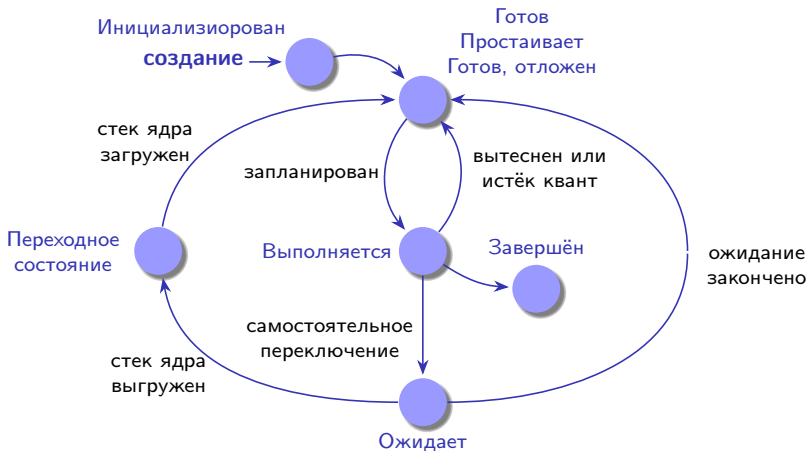
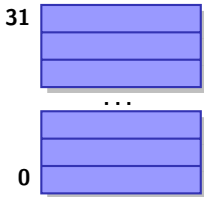


Рис. 3: граф переходов между состояниями потока

Принятие решения по переключению контекста

очереди готовых потоков



сводка готовности



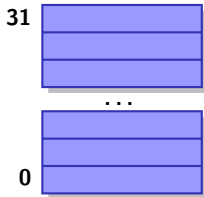
очередь отложенных

готовых потоков



CPU₀

очереди готовых потоков



сводка готовности



очередь отложенных

готовых потоков



CPU₁

Рис. 4: база данных диспетчера ядра

Вытеснение потока

Ситуации, приводящие к вытеснению

- Добровольное освобождение процессора (квант уменьшается на 1 для приоритетов < 14 , для ≥ 14 сбрасывается);
- Готовность к исполнению другого потока с более высоким приоритетом (вытесненный поток помещается в **начало** очереди готовых потоков \sim приоритета);
- Исчерпание кванта времени (приоритет может снизиться, перемещается в конец очереди, состояние «выполняется» \rightarrow «готов», вычисляется новый квант); нет потоков равного приоритета или выше \Rightarrow потоку выдаётся следующий квант;
- Завершение потока.

Поток простоя

Выполнение потока в составе System Idle Process

- Выполняется на каждом процессоре.
- Выполняется только в отсутствие других потоков (не имеет приоритета).
- Выполняет некоторые системные функции (проверяет, выбран ли какой-либо поток для выполнения на данном процессоре и организует его диспетчеризацию, вызывает процедуру из HAL для простоя, ...)

Случаи повышения приоритетов

Динамический приоритет

- После завершения операции ввода/вывода (величина определяется драйвером устройства).
- По окончании ожидания событий и семафоров (на 1 уровень).
- После выхода из состояния ожидания на объекте ядра потока с активным окном.
- После пробуждения из-за активности GUI (на 2 уровня).
- Из-за нехватки процессорного времени (инверсирование приоритета).

Предотвращение голодной смерти

Определение

Инверсирование приоритета: (*priority inversion*) — повышение до 15 динамического приоритета готовых к исполнению (состояние “ready”) потоков, ожидающих ~ 4 с.

Пример

Поток	Приоритет	Состояние
1	высокий	ожидает общего ресурса с потоком 2
2	низкий	исполняет код, блокируя ресурс
3	средний	исполняется

Предотвращение голодной смерти (окончание)

Правила работы диспетчера настройки баланса

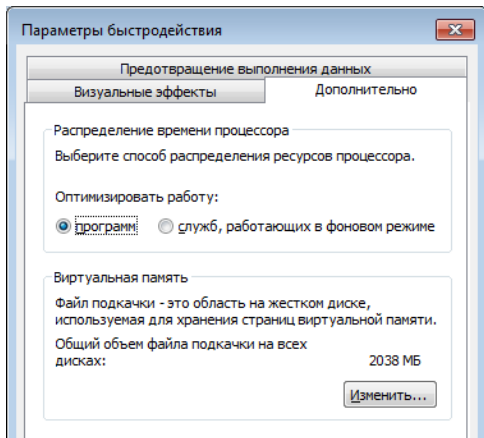
- Проверяет за каждый запуск очередные 16 готовых потоков.
- За 1 проход повышается приоритет не более 10 потоков, далее просмотр прекращается, возобновляется при следующем запуске.

Правила повышения приоритетов

Динамический приоритет

- Всегда \geq базового.
- Изначально = базовому.
- Повышение приоритета (priority boost) может выполняться для потоков с базовым приоритетом $\in [0, 15]_{\mathbb{Z}}$.
- Величина динамического приоритета также всегда $\in [0, 15]_{\mathbb{Z}}$.
- После повышения приоритет уменьшается на 1 после каждого завершения кванта времени, до понижения до базового.

Выбор кванта времени



Правила изменения квантов

При настройке (1) кванты потоков процесса с активным окном, классом приоритета выше `IDLE_PRIORITY_CLASS`, увеличиваются в соответствии с таблицей.

Рис. 5: выбор базового кванта времени ($2 \cdot 3$ или $12 \cdot 3$ тактов)

Планирование на многопроцессорных системах

Правила

- При наличии простаивающих процессоров выбирается в первую очередь идеальный, если невозможно — предыдущий, затем — тот, на котором выполняется планирование. Иначе — первый из простаивающих (с учётом Hyperthreading, NUMA).
- При отсутствии простаивающих процессоров проверяется приоритет выполняемого (running) или простаивающего (standby) потока на идеальном процессоре. Ниже \Rightarrow вытеснение.
- Иначе поток помещается в очередь готовых (ready) \sim приоритета на идеальном процессоре.
- Если очередь потоков для данного процессора пуста, запускается поток простоя, который проверяет очереди готовых потоков на других процессорах (с учётом Hyperthreading, NUMA).

Состояния процессов

Состояние	Значение
TASK_RUNNING	Выполняется или ожидает выполнения
TASK_INTERRUPTIBLE	Ожидает события (→ TASK_RUNNING)
TASK_UNINTERRUPTIBLE	Как TASK_INTERRUPTIBLE, но не реагирует на сигналы
TASK_STOPPED	Останов после некоторых сигналов
TASK_TRACED	Останов отладчиком

Таблица 6: состояния процессов, Linux

Состояние	Значение
EXIT_ZOMBIE	Завершил работу, но родитель не вызвал wait...()
EXIT_DEAD	Завершил работу, родитель вызвал wait...()

Таблица 7: дополнительные состояния процессов, Linux

Очереди процессов

Состояние	Хранение
TASK_RUNNING	организованы в 140 очередей по приоритетам (постоянное время поиска, красночёрные деревья, начиная с 2.6.23).
TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE	хранятся в очередях ожидания (не постоянное время поиска).
TASK_STOPPED, EXIT_ZOMBIE, EXIT_DEAD	не объединяются в списки.

Таблица 8: организация списков процессов

Добровольное переключение, статический приоритет

POSIX `sched_yield()` (`<sched.h>`)

```
int sched_yield(void);
```

POSIX `nice()` (`<unistd.h>`)

```
int nice(int nInc);
```

Параметр	Допустимые значения
nInc	$[-20, 19]_{\mathbb{Z}}$

Таблица 9: значения параметра

Определение политики планирования

POSIX sched_setscheduler() (<sched.h>)

```
int sched_setscheduler(
    pid_t          pid,
    int            nPolicy,
    const struct sched_param *pcParam);
```

<sched.h>

```
struct sched_param {
    /* ... */
    int sched_priority;
    /* ... */
};
```

Параметр	Допустимые значения
nPolicy	SCHED_FIFO, SCHED_RR, SCHED_OTHER
sched_priority	[0,99] _Z

Таблица 10: значения параметров

Особенности политик планирования реального времени

SCHED_FIFO

- Приоритет: $s \in [1, 99]_{\mathbb{N}}$.
- Вначале помещается в конец очереди своего приоритета.
- Прерывается процессами с бóльшим приоритетом, остаётся в начале очереди.
- Вызов `sched_yield()` перемещает в конец очереди.
- Работает до: останова, блокировки, приостанова процессом большего приоритета, вызова `sched_yield()`.

SCHED_RR

- Дополнительно: перевод в конец очереди при исчерпании кванта.

Виды процессов, планировщик $O(1)$ (2.6 ... 2.6.22)

Тип	Требования
интерактивные	быстрое пробуждение, средняя задержка $\sim 50 \div 150$ мс, малый разброс задержек
пакетные	низкий приоритет
реального времени	гарантированно малое время отклика с минимальным разбросом, неблокирование ради процессов с низшим приоритетом

Таблица 11: классификация процессов с точки зрения планировщика

Полностью справедливый планировщик ($O(\log n)$)

Алгоритм планировщика Completely Fair Scheduler (CFS, 2.6.23)

- Запланированные задачи хранятся в красно-чёрном дереве (вместо очередей), ключ — использованное процессорное время (ηs).
- При готовности к исполнению выбирается задача из самого левого узла дерева.
- Запись, соответствующая исполняемой задаче, удаляется из дерева.
- Задача исполняется в течение положенного кванта времени, после чего обновляется использованное ею процессорное время, и запись с этим ключом вновь добавляется в дерево.

Создание потока

Windows API CreateThread()

```
HANDLE WINAPI CreateThread(  
    _In_opt_ LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    _In_     SIZE_T dwStackSize,  
    _In_     LPTHREAD_START_ROUTINE lpStartAddress,  
    _In_opt_ LPVOID lpParameter,  
    _In_     DWORD dwCreationFlags, // CREATE_SUSPENDED  
    _Out_opt_ LPDWORD lpThreadId  
);
```

Создание потока (продолжение)

Пример

```
#include <windows.h>

DWORD WINAPI MyThreadProc(
    LPVOID pvData)
{
    // ...
    return 0;
}
```

Пример (окончание)

```
int main()
{
    HANDLE hThread = CreateThread(
        NULL,           // атрибуты
        0,             // стек
        MyThreadProc,  // адрес
        NULL,          // параметр
        0,             // флаги
        NULL);         // & номера
    // ...
    CloseHandle(hThread);
}
```

Создание потока (продолжение)

POSIX pthread_create()

```
int pthread_create(
    pthread_t *restrict          pThread,
    const pthread_attr_t *restrict pAttr,
    void *(*                      pStartRoutine)(void *),
    void *restrict              pArg);
```

Создание потока (окончание)

Пример

```
#include <pthread.h>

void *MyThreadProc(void *pvData)
{
    // ...
    return 0;
}
```

Пример (окончание)

```
int main()
{
    pthread_t hThread;
    int nRet = pthread_create(
        &hThread,      // & дескриптора
        NULL,          // атрибуты
        MyThreadProc, // адрес
        NULL);         // параметр
    // ...
}
```


Многопоточный параллелизм

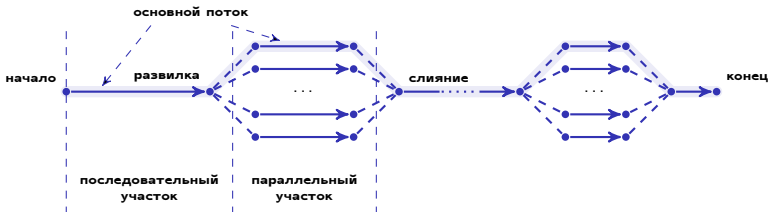


Рис. 6: концепция многопоточного параллелизма

Ожидание события

Windows API WaitForSingleObject(), WaitForMultipleObjects()

```
DWORD WINAPI WaitForSingleObject(  
    _In_     HANDLE     hHandle,  
    _In_     DWORD      dwMilliseconds // INFINITE для „∞“  
);
```

```
DWORD WINAPI WaitForMultipleObjects(  
    _In_     DWORD      nCount,  
    _In_     const HANDLE *lpHandles,  
    _In_     BOOL       bWaitAll,      // TRUE для „∀“, FALSE для „∃“  
    _In_     DWORD      dwMilliseconds  
);
```

Ожидание события (продолжение)

Windows API MsgWaitForMultipleObjects()

```
DWORD WINAPI MsgWaitForMultipleObjects(  
    _In_     DWORD    nCount,  
    _In_     const    HANDLE *pHandles,  
    _In_     BOOL     bWaitAll,  
    _In_     DWORD    dwMilliseconds,  
    _In_     DWORD    dwWakeMask           // QS_ALLINPUT, QS_MOUSE, ...  
);
```

Ожидание события (продолжение)

WAIT_OBJECT_0 ... WAIT_OBJECT_0 + nCount - 1

WAIT_OBJECT_0 + nCount

WAIT_TIMEOUT

WAIT_FAILED

Таблица 12: значения, возвращаемые функциями ...WaitFor...()

Ожидание события (окончание)

Пример (Windows API WaitForSingleObject())

```
int main()
{
    HANDLE hThread = CreateThread(
        NULL, 0, MyThreadProc, NULL, 0, NULL);
    // ...
    WaitForSingleObject(hThread, INFINITE);
    // ...
}
```

Пример ожидания события

Пример (Windows API WaitForMultipleObjects())

```
int main()
{
    int i;
    HANDLE ahThreads[MY_NUM_THREADS];
    for (i = 0; i < MY_NUM_THREADS; ++ i)
        ahThreads[i] = CreateThread(
            NULL, 0, MyThreadProc, NULL, CREATE_SUSPENDED, NULL);
    for (i = 0; i < MY_NUM_THREADS; ++ i)
        ResumeThread(ahThreads[i]);
    // ...
    WaitForMultipleObjects(
        MY_NUM_THREADS, ahThreads, TRUE, INFINITE);
}
```

Пример ожидания события (продолжение)

POSIX pthread_join()

```
int pthread_join(  
    pthread_t thread,  
    void ** pValue);
```

Пример ожидания события (окончание)

Пример (POSIX pthread_join())

```
int main()
{
    int i;
    pthread_t ahThreads[MY_NUM_THREADS];
    for (i = 0; i < MY_NUM_THREADS; ++ i)
        pthread_create(
            &ahThreads[i], NULL, MyThreadProc, NULL);
    // ...
    for (i = 0; i < MY_NUM_THREADS; ++ i)
        pthread_join(ahThreads[i], NULL);
    // ...
}
```


Пример: сумма векторов POSIX

Пример

```
#include <pthread.h>

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
```

Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
void get_data(size_t *puDataSize, double **ppdData)
{
    const size_t cuDataSize = 1000013;
    size_t i;
    *puDataSize = cuDataSize;
    *ppdData = (double *) calloc(cuDataSize, sizeof (double));
    for (i = 0; i < cuDataSize; ++ i)
        (*ppdData)[i] = 1;
}
```

Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
void free_data(double *pdData)
{
    free(pdData);
}
```

Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
double sum(double *pdData, size_t uStart, size_t uEnd)
{
    size_t i;
    double dResult = 0;
    //
    for (i = uStart; i < uEnd; ++ i)
        dResult += pdData[i];
    //
    return dResult;
}
```

Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
typedef
struct
{
    size_t m_uStart, m_uEnd;
    double *m_pdData;
    double m_dResult;
}
ThreadParams;
```

Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
void *thread_proc(void *pvParam)
{
    ThreadParams *pParams = (ThreadParams *) pvParam;
    pParams->m_dResult = sum(
        pParams->m_pdData, pParams->m_uStart, pParams->m_uEnd);
    return NULL;
}
```

Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
int main()
{
    size_t i, uDataSize;
    const size_t cuNumThreads = 8;
    pthread_t ahThreads[cuNumThreads - 1];
    ThreadParams aParams[cuNumThreads];
    double *pdData;
    double dResult = 0;
    //
    get_data(&uDataSize, &pdData);
    //
```

Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
size_t uChunkSize = uDataSize / cuNumThreads;
size_t uIndex = 0, uPrev = 0;
for (i = 0; i < cuNumThreads; ++ i)
{
    if (i == cuNumThreads - 1)
        uIndex = uDataSize;
    else
        uIndex += uChunkSize;
    //
```


Пример: сумма векторов POSIX (продолжение)

Пример (продолжение)

```
aParams[i].m_uStart = uPrev;
aParams[i].m_uEnd = uIndex;
aParams[i].m_pdData = pdData;
uPrev = uIndex;
//
if (i < cuNumThreads - 1)
    pthread_create(&ahThreads[i], NULL, &thread_proc, &aParams[i]);
} // for (i = 0; i < cuNumThreads; ++ i)
//
thread_proc(&aParams[cuNumThreads - 1]);
//
```

Пример: сумма векторов POSIX (окончание)

Пример (окончание)

```
for (i = 0; i < cuNumThreads - 1; ++ i)
    pthread_join(ahThreads[i], NULL);
//
for (i = 0; i < cuNumThreads; ++ i)
    dResult += aParams[i].m_dResult;
//
printf("Result: %lf\n", dResult);
//
free_data(pdData);
//
} // main()
```

Сборка и запуск примера

Пример

```
$ gcc -pthread vec_sum_posix.c  
$ ./a.out  
Result: 1000013.000000
```