Многопоточное программирование с использованием PThreads

PTHREADS

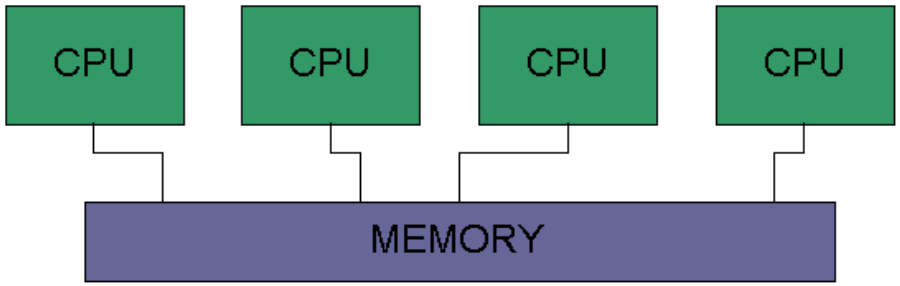
- **PTHREADS** – переносимая библиотека для разработки многопоточных программ
- Стандартизована POSIX Section 1003.1c
- Дает возможность получить эффективность близкую к максимально-возможной, но при этом требуется серьезная переработка кода

Источники информации

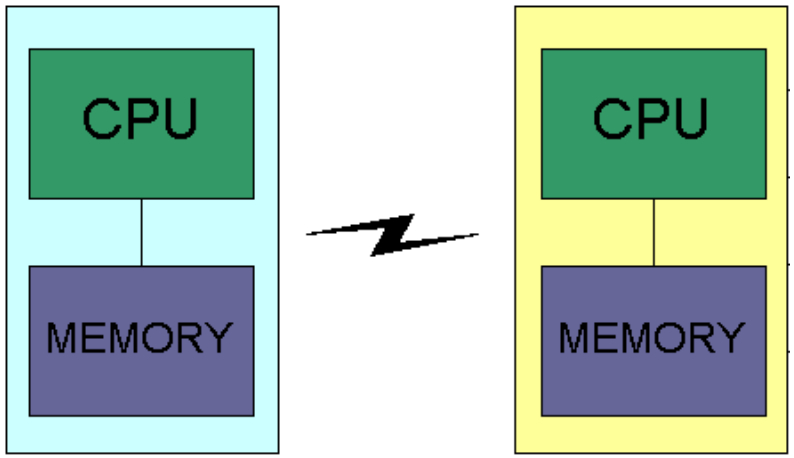
- POSIX Section 1003.1c
- http://pubs.opengroup.org/onlinepubs/9699919799/
- ***Pthreads Programming A POSIX Standard for Better Multiprocessing*** Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell O'Reilly
- ***Programming with POSIX(R) Threads*** David R. Butenhof

Виды параллелизма

Общая память



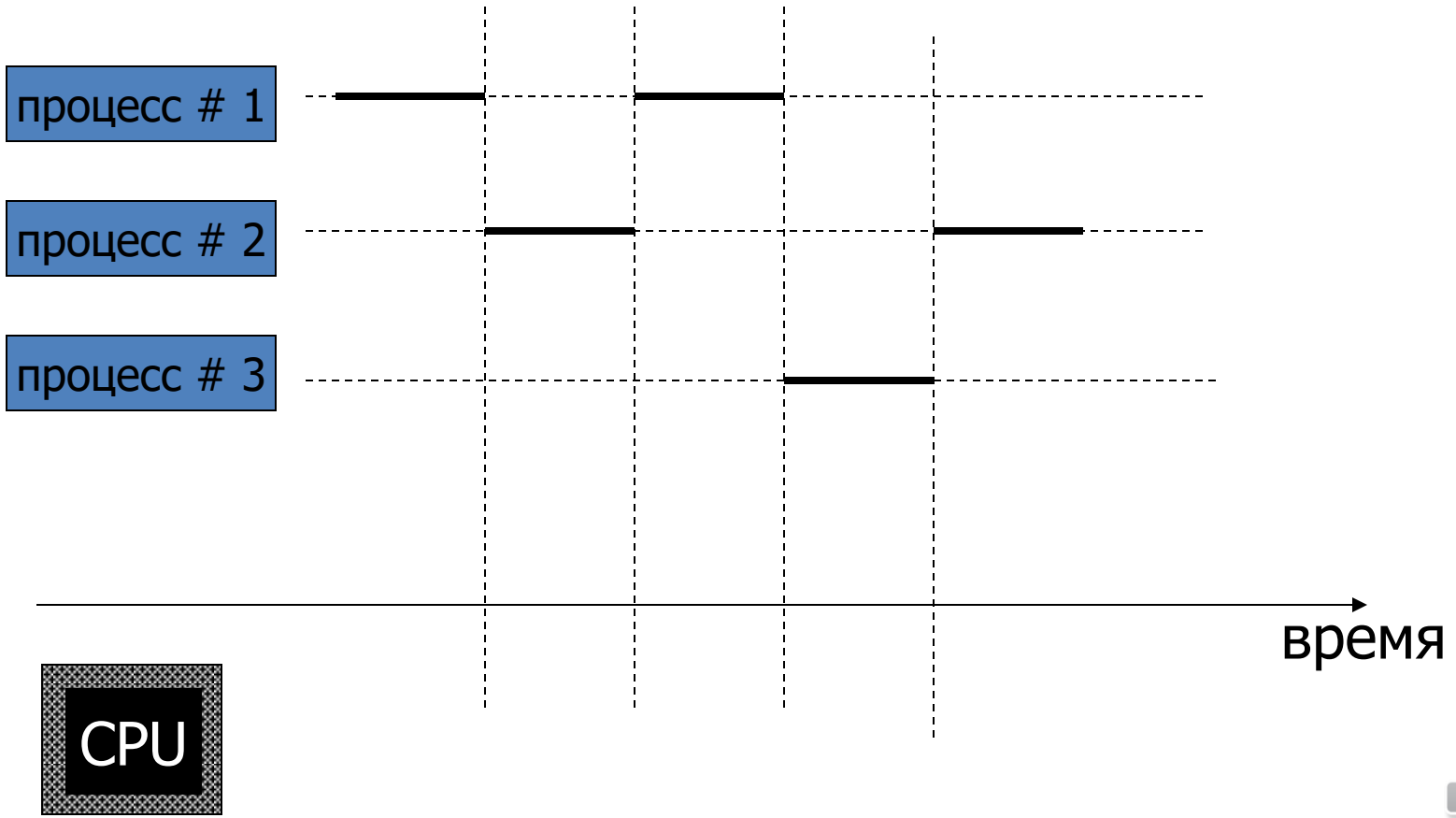
Распределенная память



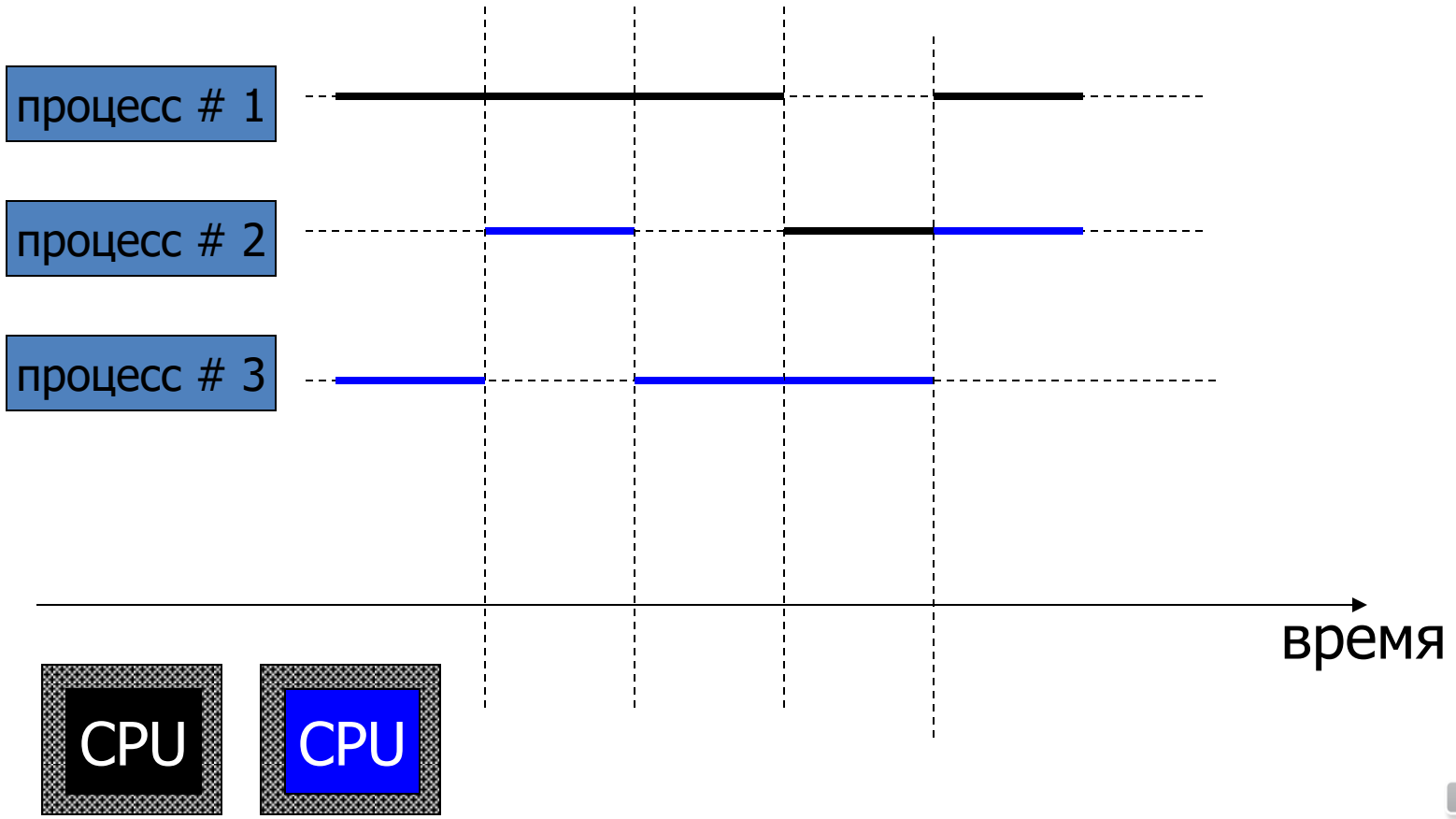
ПРОЦЕССЫ В ОПЕРАЦИОННОЙ СИСТЕМЕ

- **Процесс** – это экземпляр выполняемой программы.
- Контекст:
  - регистры;
  - таблица трансляции адресов памяти;
  - ...
- Адресное пространство:
  - код программы;
  - статические данные;
  - стек;
  - разделяемая память;
  - динамическая память (куча).

ПЛАНИРОВАНИЕ ПРОЦЕССОВ



ПЛАНИРОВНАИЕ ПРОЦЕССОВ В МНОГОПРОЦЕССОРНОЙ СИСТЕМЕ

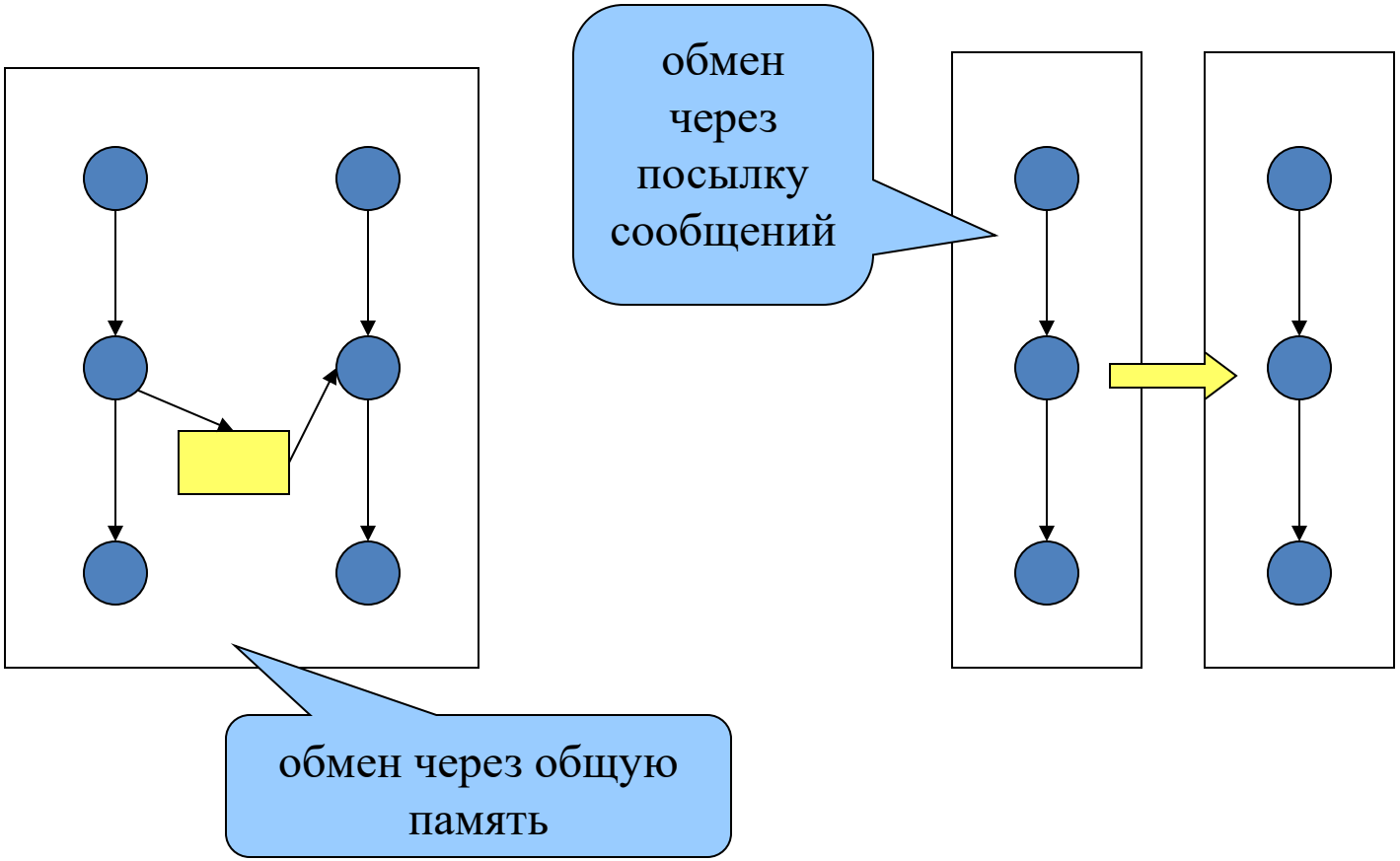


ТРЕДЫ

Тредами (*потоки, нити)* называются параллельно выполняющиеся потоки управления в рамках одного процесса.

Треды одного процесса разделяют его адресное пространство.

Треды и процессы



Различие тредов и процессов

- Различные треды выполняются в одном адресном пространстве.
- Различные процессы выполняются в разных адресных пространствах.
- Треды имеют «собственный» стек и набор регистров. Глобальные данные являются общими.
- Как локальные, так и глобальные переменные процессов являются «собственными».

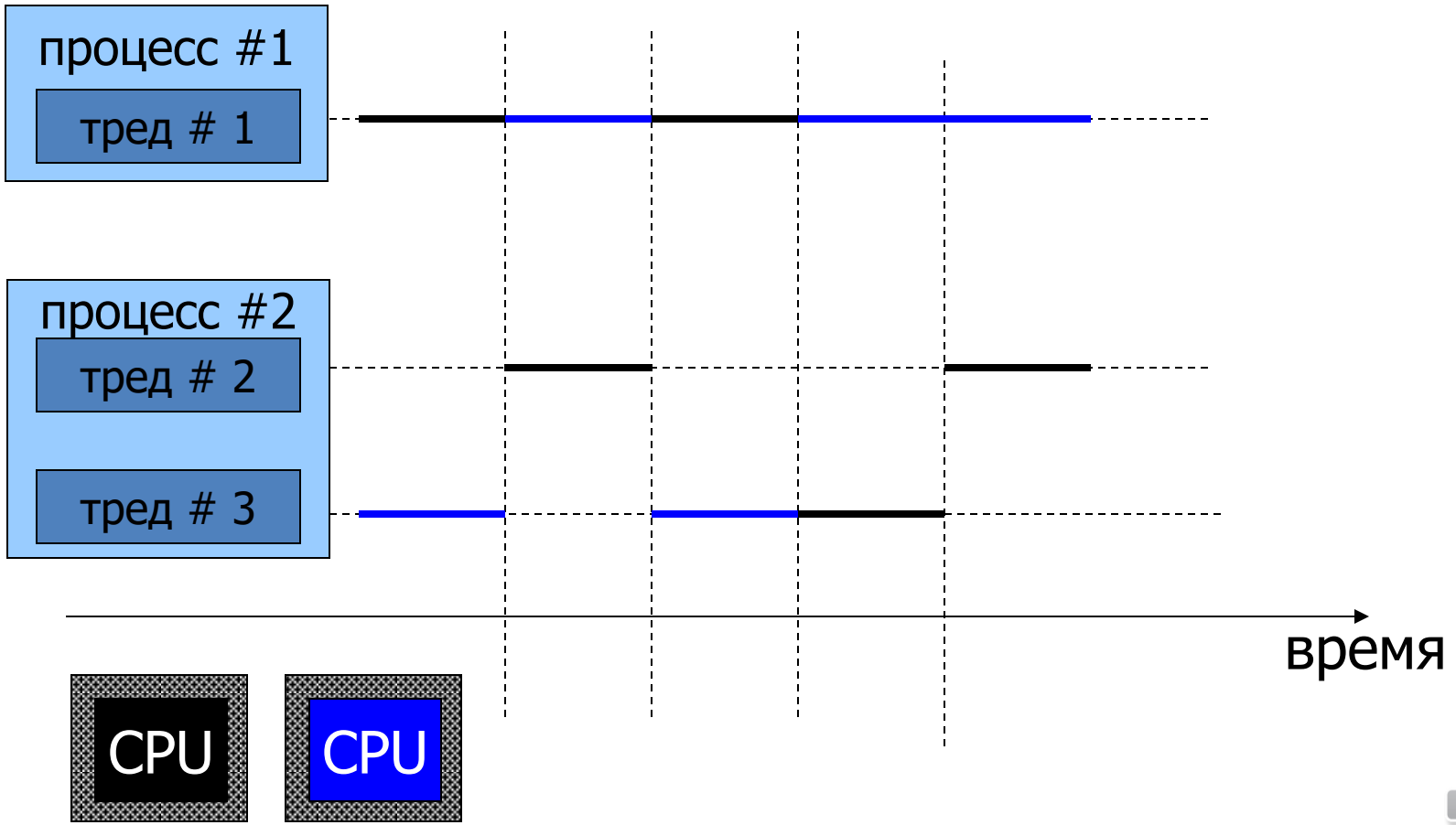
Реализация потоков

- Пользовательские потоки (не используются механизмы ядра)
  Такие потоки не «видны» ОС и поэтому не могут выполняться разными ядрами, если принадлежат одному процессу
- Потоки ядра (потоки ОС)
  Такие потоки могут использовать преимущества многоядерности, но их переключение требует бОльших ресурсов, чем пользовательских

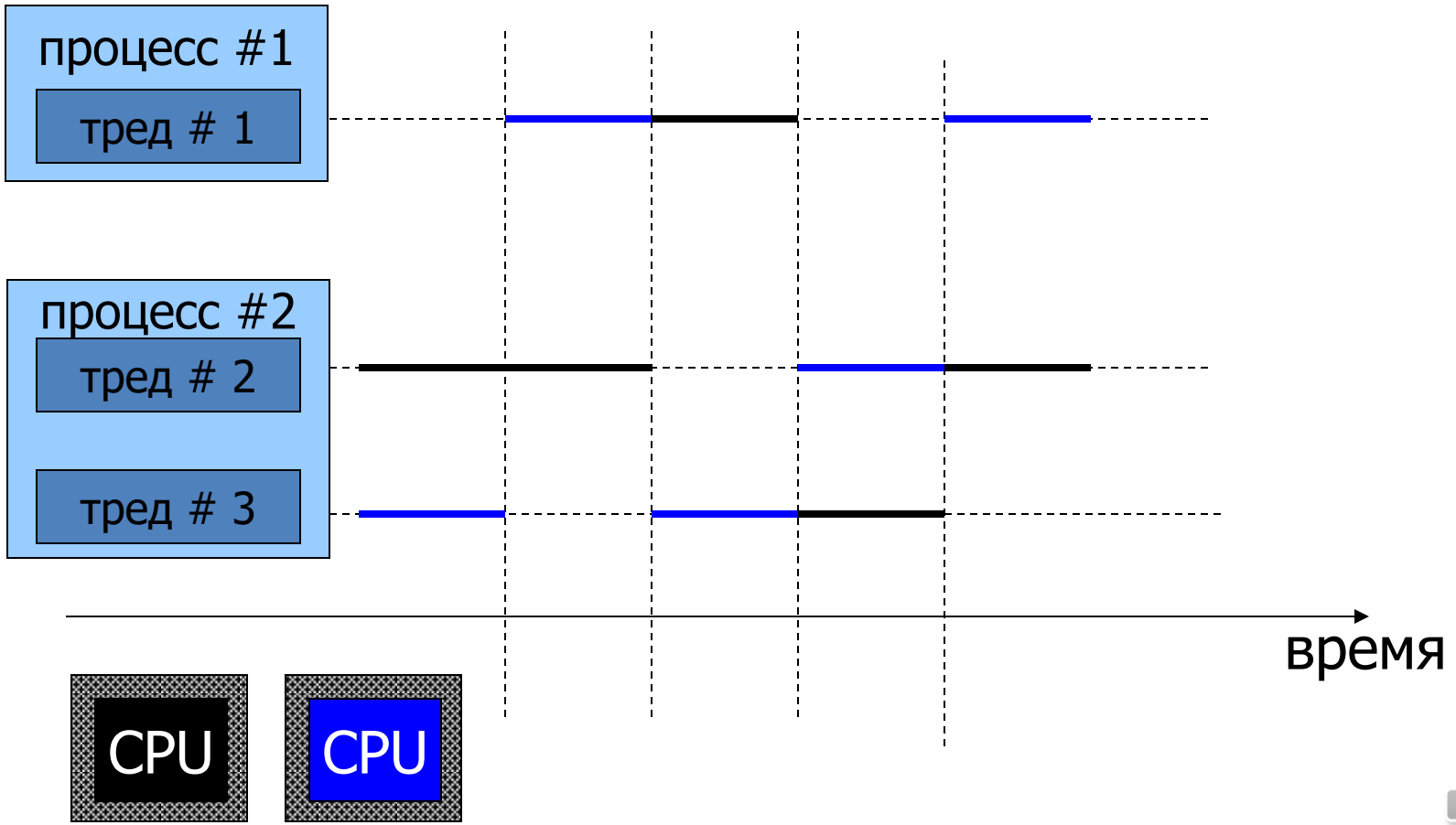
Реализация потоков пользователя

- Используются вызовы makecontext, swapcontext, позволяющие переключать контекст исполнения (косвенный вызов функции)
- Для передачи управления между потоками используется либо механизм явной передачи (специальный вызов, предусмотренный библиотекой потоков), либо механизм сигналов ОС

ПЛАНИРОВАНИЕ ТРЕДОВ (УРОВЕНЬ ПОЛЬЗОВАТЕЛЯ)



ПЛАНИРОВАНИЕ ТРЕДОВ (УРОВНЯ ЯДРА)

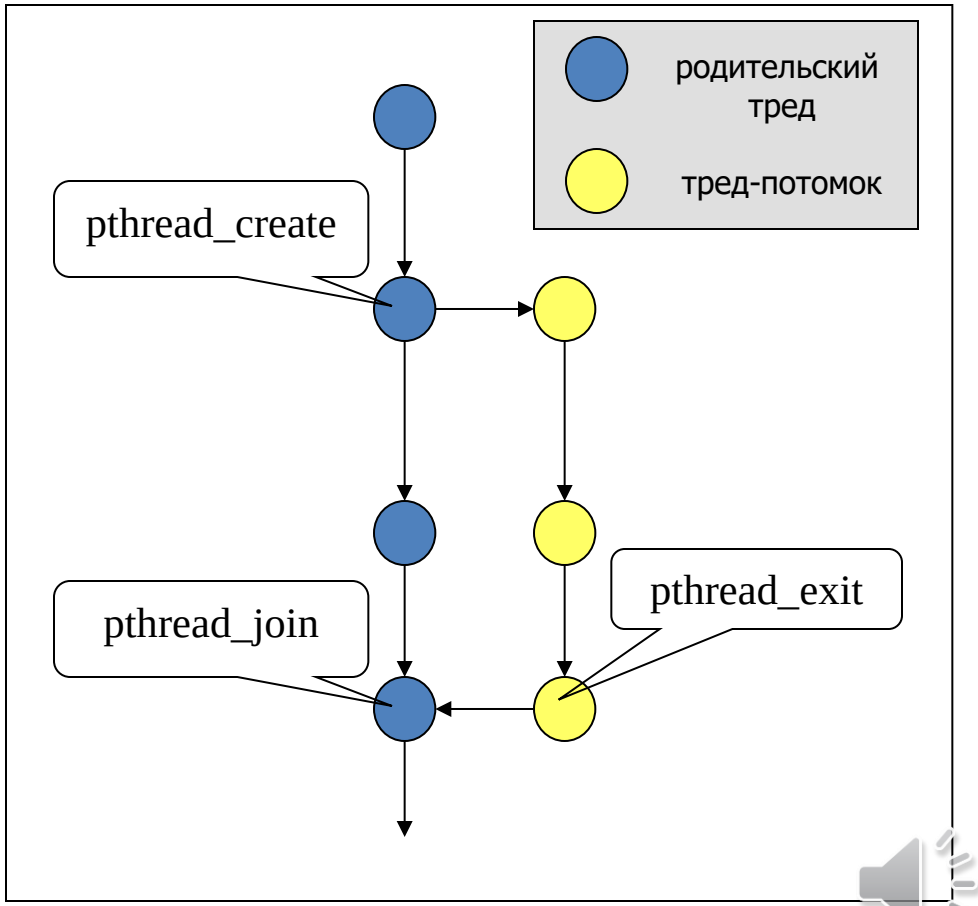


ГИБРИДНАЯ РЕАЛИЗАЦИЯ

При гибридной реализации несколько пользовательских потоков отображаются на один поток ядра. Это позволяет с одной стороны, использовать многоядерность в рамках одного процесса, а с другой, при большом числе потоков не приводит к слишком большим накладным расходам, связанным с переключением потоков ядра.

Создание и завершение тредов

```
int pthread_create (
  pthread_t * outHandle,
  pthread_attr_t *inAttribute,
  void *(*inFunction)(void *),
  void *inArg
);

void pthread_exit(void *inReturnValue)

int pthread_join(
    pthread_t inHandle,
    void **outReturnValue,
);
```



Создание треда

int pthread_create (pthread_t * outHandle, pthread_attr_t *inAttribute,
  void *(*inFunction)(void *), void *inArg);

outHandle – используется для возвращение в тред-родитель идентификатора треда потомка;
inAttribute – атрибуты треда;
inFunction – указатель на функцию, содержащую код, выполняемый тредом;
inArg – указатель на аргумент, передаваемый в тред;

Завершение треда

void pthread_exit(void *inReturnValue)

Вызов этой функции приводит к завершению треда. Процесс-родитель получает указатель в качестве возвращаемых данных.

Обычное завершение функции и возврат указателя на void*, выполняемой тредом эквивалентно вызову функции pthread_exit, которая используется в случае, когда надо завершить тред из функций, вызванных этой функцией.

Обработка завершения треда на треде-родителе

int pthread_join(pthread_t inHandle, void **outReturnValue);

Вызов этой функции приводит к блокировке родительского треда до момента завершения треда-потомка, соответствующего индентификатору inHandle. В область, указанную параметром outReturnValue, записывается указатель, возвращенный завершившимся тредом.

pthread_join приводит к освобождению ресурсов, отведенных на тред (в частности сохранненого возращаемого значения). Необходимо выполнять также для синхронизации основного треда и тредов-потомков.

Возвращаемый статус потока

- Потоки с возвращаемым статус (Joinable threads) – результат выполнения потока сохраняется в системе и освобождается только при соответствующем вызове pthread_join.
- Потоки без возвращаемого статуса (Detached threads) – результат выполнения потока не сохраняется и не может быть получен с помощью pthread_join.

Создание потока без сохранения статуса с помощью атрибута

```
pthread_attr_t detached_attr;
pthread_attr_init(&detached_attr);
pthread_attr_setdetachedstate(&detached_attr,
PTHREAD_CREATE_DETACHED);
pthread_create(&thread, &detached_attr, ...);
pthread_attr_destroy(&detached_attr);
```

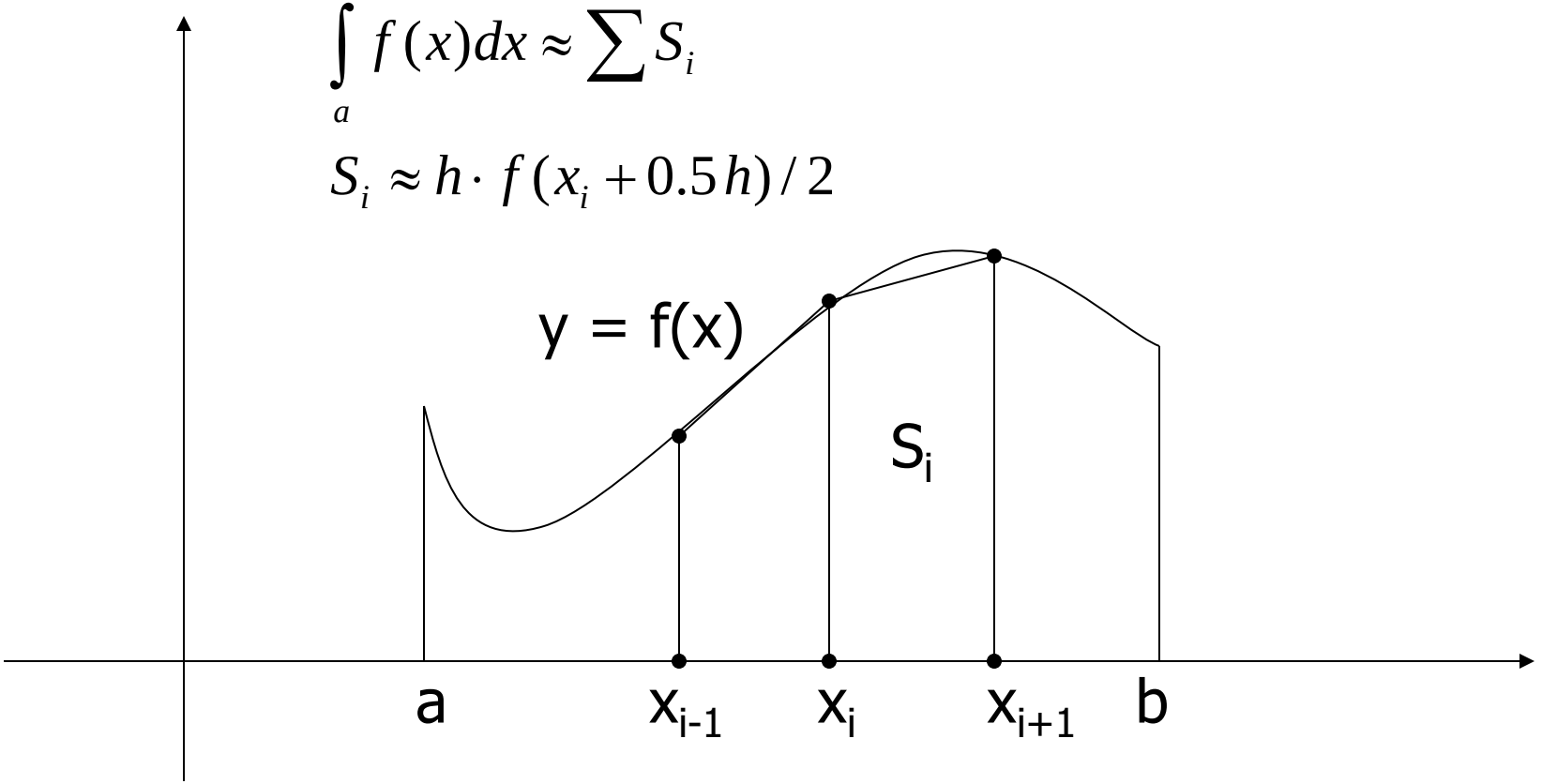
Функция **pthread_attr_setdetachedstate** устанавливает значение атрибута сохранения статуса либо **PTHREAD_CREATE_DETACHED** либо **PTHREAD_CREATE_ JOINABLE**

Функция **pthread_attr_getdetachedstate** возвращает текущую установку атрибута

Пример: вычисление определенного интеграла

$$\int_a^b f(x)dx \approx \sum S_i$$

$$S_i \approx h \cdot f(x_i + 0.5h)/2$$



$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

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

double a = 0.0, b = 1.0, h, *r;
int *nums, numt, n;

double f(double x)
{
  return 4 / (1 + x * x);
}
```

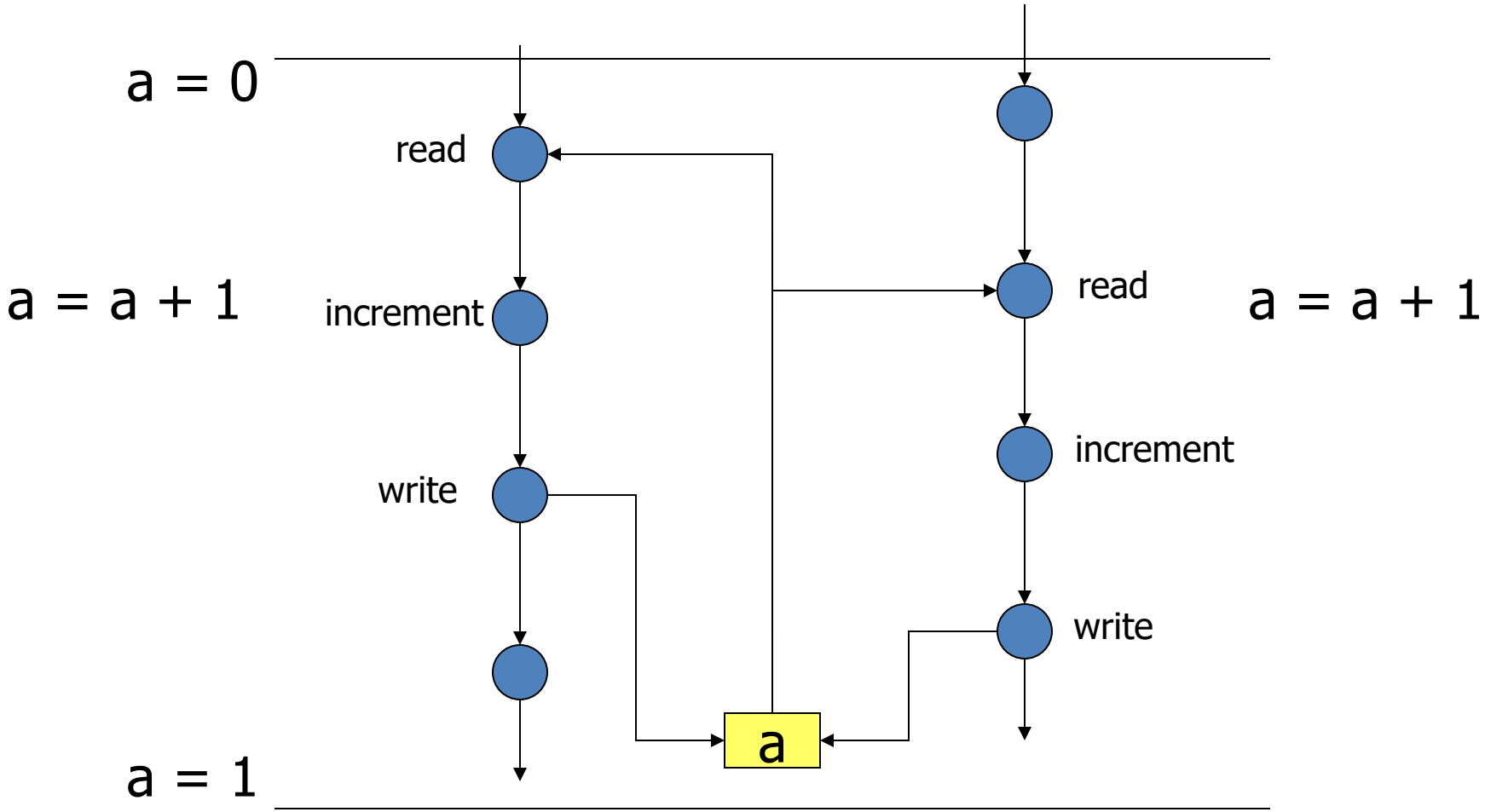
```
void* worker(void* p)
{
  int my, i;
  double s;

  my = *(int*)p;
  s = 0.0;
  for(i = my; i < n; i += numt)
    s += f(i * h + 0.5 * h);
  r[my] = s;
  return NULL;
}
```

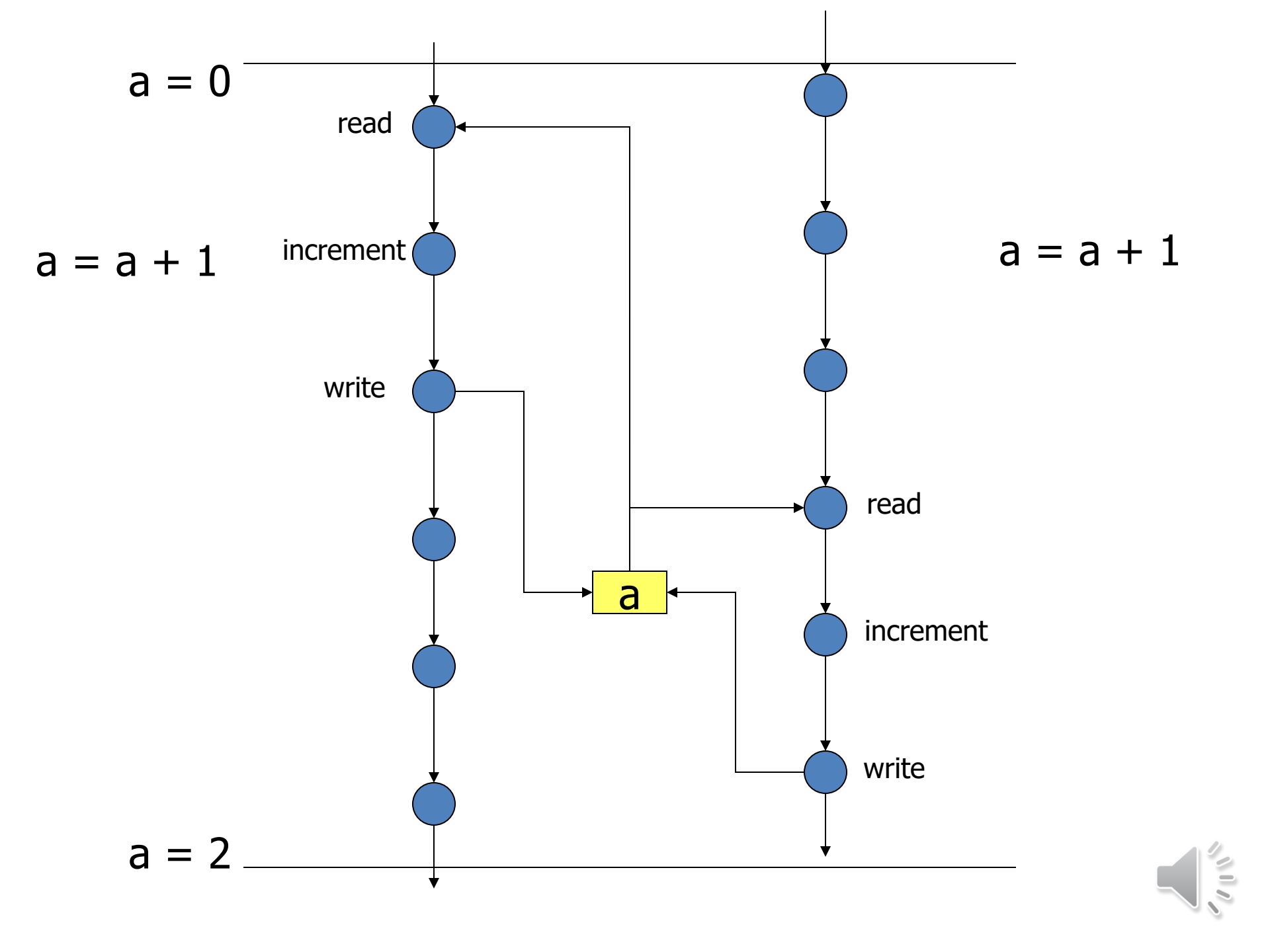
```
main(int arc, char* argv[])
{
  double S;
  pthread_t *threads;
  int i, rc;

  numt = atoi(argv[1]);
  n = atoi(argv[2]);
  threads = (pthread_t*)malloc(numt * sizeof(pthread_t));
  nums = (int*)malloc(numt * sizeof(int));
  r = (double*)malloc(numt * sizeof(double));
  h = (b - a) / n;

  for(i = 0; i < numt; i ++) {
    nums[i] = i;

    rc = pthread_create(threads + i, NULL, worker, nums + i);
    if(rc != 0) {
      fprintf(stderr, "pthread_create: error code %d\n", rc);
      exit(-1);
    }
  }
```

```
for(i = 0; i < numt; i ++) {
    rc = pthread_join(threads[i], NULL);
    if(rc != 0) {
      fprintf(stderr, "pthread_join: error code %d\n", rc);
      exit(-1);
    }
  }

  S = 0;
  for(i = 0; i < numt; i ++)
    S += r[i];
  printf("pi = %lf\n", S * h);
}
```

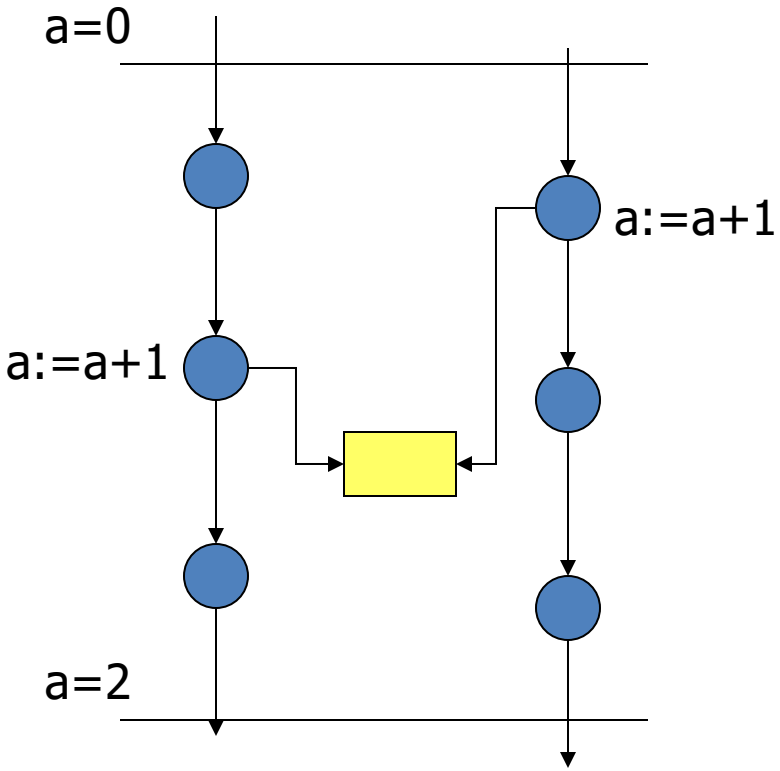
Проблема недетерминизма

Программа называется недетерминированной, если при одних и тех же входных данных она может демонстрировать различное наблюдаемое поведение


a = 0
a = a + 1
read
increment
write
read
increment
write
a = a + 1
a
a = 1


a = 0
read
a = a + 1
increment
a = a + 1
write
read
a
increment
write
a = 2

Неделимая операция



Неделимой называется операция, в момент выполнения которой состояние общих переменных не может «наблюдаться» другими тредами

Семафоры

Семафорами называются общие переменные, которые принимают неотрицательные значение целого типа для работы с которыми предусмотрены две **неделимые** операции:

1) увеличить значение семафора на 1;
2) дождаться пока значение семафора не станет положительным и уменьшить значение семафора на 1.

Поддержка семафоров в библиотеке pthreads

sem_t – тип семафора

sem_init(sem_t* semaphor, int flag, int value)

semaphor – семафор,

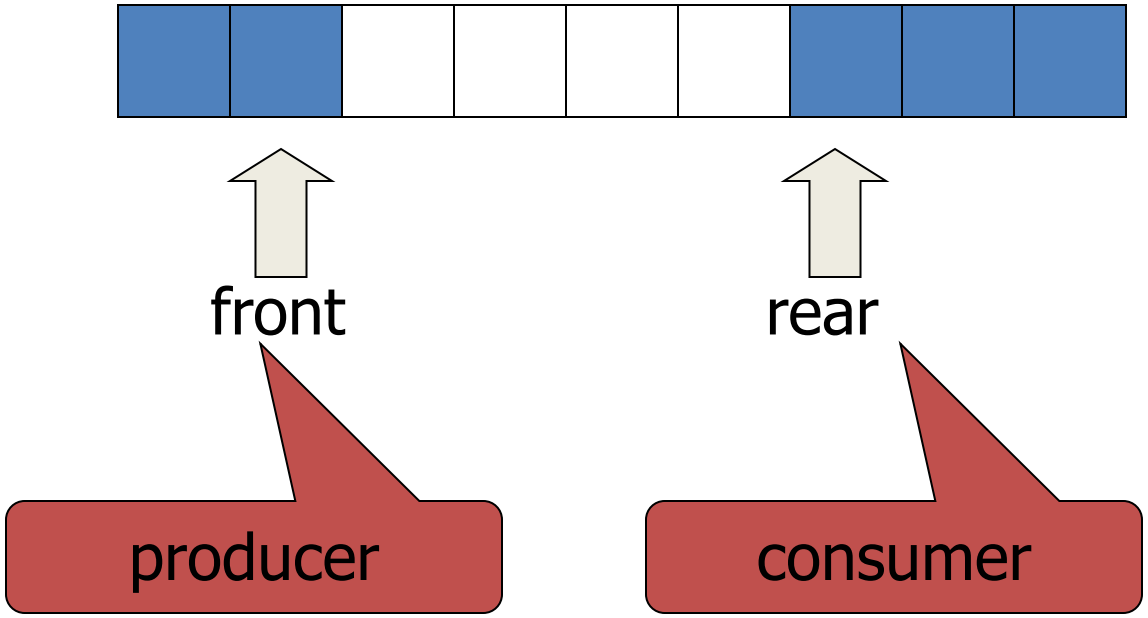
flag – флаг (0 – внутри процесса, 1 – между процессами)

value – начальное значение

sem_post(sem_t* semaphor) – увеличение семафора

sem_wait(sem_t* semaphor) – уменьшение семафора

Кольцевой буфер



```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 3

static int buf[N];

static int rear;
static int front;
static sem_t empty;
static sem_t full;


void
init ()
{
  front = 0;
  rear = 0;
  sem_init (&empty, 0, N);
  sem_init (&full, 0, 0);
}
```

```
void process(int number)
{
  sleep(number);
}

void *
consumer (void *arg)
{
  int i = 0;

  while (i != -1)
    {
      sem_wait (&full);
      i = buf[rear];
      process(i);
      printf ("consumed: %d\n", i);
      rear = (rear + 1) % N;
      sem_post (&empty);
    }
}
```

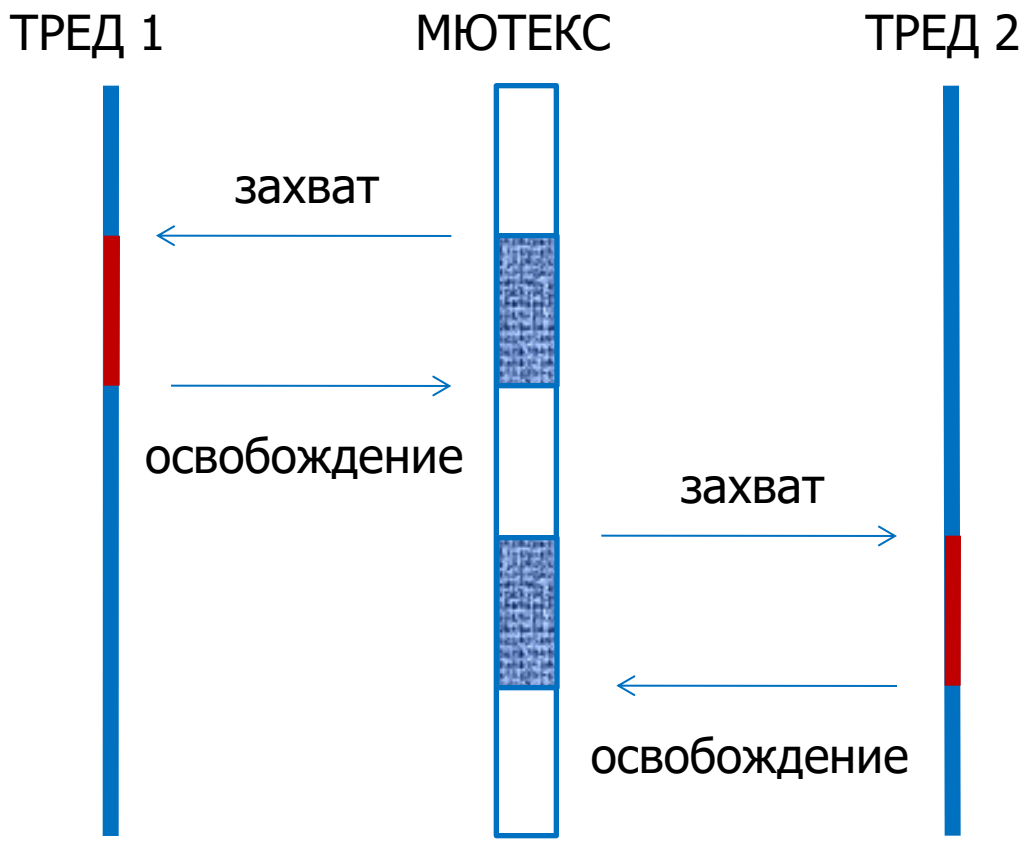
```
void *
producer (void *arg)
{
  int i;

  i = 0;
  while (i != -1)
    {
      sem_wait (&empty);
      printf ("Enter number:");
      scanf ("%d", &i);
      buf[front] = i;
      front = (front + 1) % N;
      sem_post (&full);
    }
}
```

```
main (int argc, char *argv[])
{
  pthread_t pt;
  pthread_t ct;

  init ();
  pthread_create (&pt, NULL, producer, NULL);
  pthread_create (&ct, NULL, consumer, NULL);
  pthread_join (ct, NULL);
  pthread_join (pt, NULL);
}
```

Критические секции

Критической секцией называется фрагмент кода программы, который может одновременно выполнятся только одним тредом.

Реализация механизма критических секций



Поддержка критических секций в pthreads

«Мютекс» - mutex – mutual exclusion (взаимное исключение);

Объявление и инициализация:

pthread_mutex_t – тип для взаимного исключения;

pthread_mutex_init(pthread_mutex_t* mutex, void* attribute);

pthread_mutex_destroy(pthread_mutex_t* mutex);

Захват и освобождение мютекса:

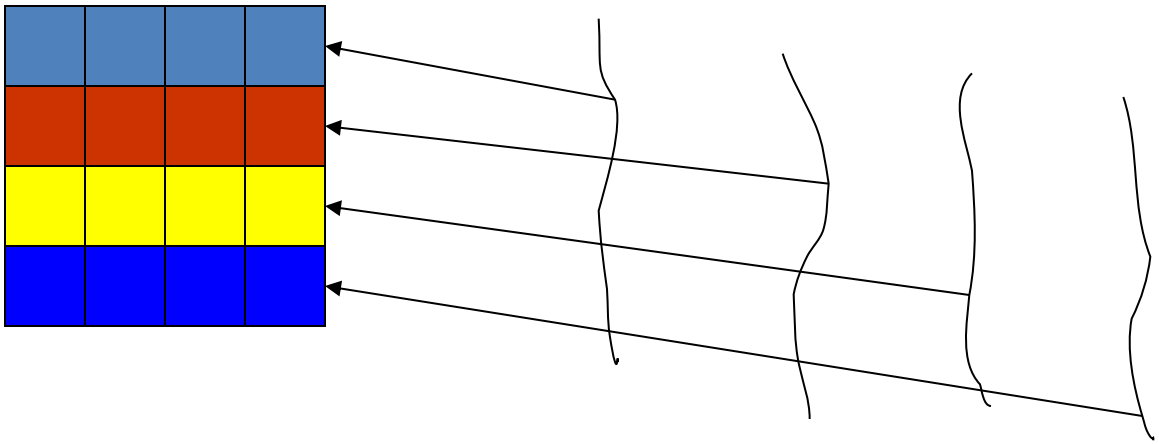
pthread_mutex_lock(pthread_mutex_t* mutex);

pthread_mutex_unlock(pthread_mutex_t* lock);

Освобождение мютекса может быть осуществлено только тем тредом, который производил его захват.

Пример: умножение матриц

C



C = A * B

каждый тред вычисляет свою строку матрицы

Умножение матриц: код программы

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <windows.h>


pthread_mutex_t mut;
static int N, nrow;
static double *A, *B, *C;
```

```
void
setup_matrices ()
{
  int i, j;

  A = (double*)malloc (N * N * sizeof (double));
  B = (double*)malloc (N * N * sizeof (double));
  C = (double*)malloc (N * N * sizeof (double));

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      {
            A[i * N + j] = 1;
            B[i * N + j] = 2;
      }
}

void
print_result ()
{
…
}
```

```
void * worker (void *arg)
{
  int i;
  while (1)
    {
      int oldrow;

      pthread_mutex_lock (&mut);
              if(nrow >= N) {
                        pthread_mutex_unlock (&mut);
                        break;
              }
      oldrow = nrow;
      nrow++;
      pthread_mutex_unlock (&mut);
```

```
for (i = 0; i < N; i++)
            {
              int j;
              double t = 0.0;

              for (j = 0; j < N; j++)
                t += A[oldrow * N + j] * B[j * N + i];

              C[oldrow * N + i] = t;
            }
    }
  return NULL;
}
```

```
void main (int argc, char *argv[])
{
  DWORD start, end;
  int i, nthreads;
  pthread_t *threads;
  pthread_mutex_init(&mut, NULL);
  nthreads = atoi (argv[1]);
  threads = (pthread_t*) malloc (nthreads * sizeof (pthread_t));

  N = atoi (argv[2]);
  setup_matrices ();
  start = GetTickCount();
for (i = 0; i < nthreads; i++)
    pthread_create (threads + i, NULL, worker, NULL);

  for (i = 0; i < nthreads; i++)
    pthread_join (threads[i], NULL);
  end = GetTickCount() - start;
  if (argc > 3)
    print_result ();
  printf("%d ms\n", end);
  pthread_mutex_destroy(&mut);

}
```

Условные переменные

Условные переменные – специальные переменные, которые служат для синхронизации и передачи сигналов между потоками.

Реализация барьерной синхронизации (мьютексы)

```
void barrier() {
  pthread_mutex_lock(&bar);
  num ++;
  pthread_mutex_unlock(&bar);

  while(1) {
    if(num == N) break;
  }
}
```

проблема – неэффективное использование процессорного ресурса

Реализация барьерной синхронизации (мьютексы)

```
void barrier() {
  pthread_mutex_lock(&bar);
  num ++;
  pthread_mutex_unlock(&bar);

  while(1) {
    if(num == N) break;
    sleep(1);
  }
}
```

проблема – ненужные ожидания

Нужен механизм сигналов!!!

Инициализация условной переменной в pthread

int **pthread_cond_init**(pthread_cond_t* cv,
pthread_cond_attr_t* attr);

cv – указатель на инициализируемую условную переменную

attr – атрибуты (по умолчанию NULL)

Освобождение условной переменной в pthread

int **pthread_cond_destroy**(pthread_cond_t* cv);

cv – указатель на освобождаемую условную переменную

Ожидание на условной переменной в pthread

int **pthread_cond_wait**(pthread_cond_t* cv pthread_mutex_t* m)

Освобождает мьютекс и переходит в режим ожидания (всегда вызывается с заблокированным мьютексом **m**);

cv – указатель на условную переменную;

m – указатель на мьютекс, соответствующий критической секции;

по завершению – захватывает мьютекс **m**.

Семантика ожидания

- Операция **pthread_cond_wait** производит динамическое связывание мьютекса и условной переменной.
- Нельзя одновременно связывать разные мьютексы с одной и той же условной переменной.

Сигнализация по условной переменной в pthread

int pthread_cond_signal(pthread_cond_t* cv);
разблокирует как минимум один поток, ожидающий условную переменную;

int pthread_cond_broadcast(pthread_cond_t* cv);
разблокирует все потоки, ожидающие переменные;

cv – указатель на условную переменную;

Корректность работы с условными переменными

- Все функции для работы с условными переменными должны вызываться из критических секций с одним и тем же мьютексом
- Это необходимо, т.к. в противном случае будет нарушена корректность работы: одновременная модификация и чтение общей условной переменной

Реализация барьерной синхронизации при помощи условных переменных

```
void barrier() {
  pthread_mutex_lock(&bar);
  num ++;
  if(num < N)
    pthread_cond_wait(&go, &bar);
  else {
    num = 0;
    pthread_cond_broadcast(&go);
  }
  pthread_mutex_unlock(&bar);
}
```


Реализация Метода Гаусса с помощью условных переменных

ПОСЛЕДОВАТЕЛЬНЫЙ ВАРИАНТ

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>

#define N 1000

double A[N][N], B[N], X[N];

void init() {
  int i, j;

  for(i = 0; i < N; i ++) {
    for(j = 0; j < N; j ++) {
      A[i][j] = 1.;
    }
    A[i][i] = 10.;
    B[i] = N + 9.;
  }
}
```

```
void printres() {
  int i;

  for(i = 0; i < N; i ++) {
    printf("%lf ", X[i]);
  }
  printf("\n");
}
```

```
main() {
  init();
  gauss();
  printres();
}
```

```
void gauss() {
  int i, j, k, si;
  double maxnorm, e;
  for(i = 0; i < (N - 1); i ++) {
    maxnorm = 0.;
    for(j = i; j < N; j ++) {
      e = fabs(A[j][i]);
      if(e > maxnorm){
        maxnorm = e;
        si = j;
      }
    }
    if(maxnorm == 0.) {
      fprintf(stderr, "Singular matrix!\n");
      exit(-1);
    }
    if(si != i) {
      e = B[i];
      B[i] = B[si];
      B[si] = e;
      for(j = i; j < N; j ++) {
        e = A[i][j];  A[i][j] = A[si][j]; A[si][j] = e;
      }
    }
    for(j = i + 1; j < N; j ++) {
        e = A[j][i] / A[i][i];
        B[j] -= e * B[i];
        for(k = i; k < N; k ++) {
          A[j][k] -= e * A[i][k];
        }
      }
    }
    for(i = N - 1; i >= 0; i --) {
      X[i] = B[i];
      for(j = N - 1; j > i; j --) {
        X[i] -= X[j] * A[i][j];
      }
      X[i] /= A[i][i];
    }
}
```

ПАРАЛЛЕЛЬНЫЙ ВАРИАНТ: ФУНКЦИЯ **main**

```
double A[N][N], B[N], X[N];
pthread_mutex_t mut, mute;
pthread_cond_t cv, cve;
int i, counter, tcnt = 0, numt, dogauss = 0;

main() {
  pthread_t mthread;
  pthread_t* threads;
  int j;
  numt = 4;
  threads = (pthread_t*)malloc(numt * sizeof(pthread_t));
  pthread_mutex_init(&mut, NULL);
  pthread_mutex_init(&mute, NULL);
  pthread_cond_init(&cv, NULL);
  pthread_cond_init(&cve, NULL);
  init();
```

ПАРАЛЛЕЛЬНЫЙ ВАРИАНТ: ФУНКЦИЯ **main**

```
pthread_create(&mthread, NULL, gaussm, NULL);
  for(j = 0; j < numt; j ++) {
    pthread_create(threads + j, NULL, gauss, NULL);
  }
  for(j = 0; j < numt; j ++) {
    pthread_join(threads[j], NULL);
  }
  pthread_join(mthread, NULL);
  pthread_cond_destroy(&cv);
  pthread_cond_destroy(&cve);
  pthread_mutex_destroy(&mut);
  pthread_mutex_destroy(&mute);
  free(threads);
}
```

ПАРАЛЛЕЛЬНЫЙ ВАРИАНТ: УПРАВЛЯЮЩИЙ ПОТОК

```
void* gaussm(void* arg) {
  int j, k, si;
  double maxnorm, e;

  for(i = 0; i < (N - 1); i ++) {
    maxnorm = 0.;
    for(j = i; j < N; j ++) {
      e = fabs(A[j][i]);
      if(e > maxnorm){
        maxnorm = e;
        si = j;
      }
    }
    if(maxnorm == 0.) {
      fprintf(stderr, "Singular matrix!\n");
      exit(-1);
    }
```

```
if(si != i) {
      e = B[i];
      B[i] = B[si];
      B[si] = e;
      for(j = i; j < N; j ++) {
        e = A[i][j];
        A[i][j] = A[si][j];
        A[si][j] = e;
      }
    }
```

ПАРАЛЛЕЛЬНЫЙ ВАРИАНТ: УПРАВЛЯЮЩИЙ ПОТОК

```
    pthread_mutex_lock(&mut);
    counter = 0;
    dogauss = 1;
    pthread_cond_broadcast(&cv);
    pthread_mutex_unlock(&mut);

    pthread_mutex_lock(&mute);
    while(counter != numt) {
      pthread_cond_wait(&cve, &mute);
    }
    pthread_mutex_unlock(&mute);
  }
  for(i = N - 1; i >= 0; i --) {
    X[i] = B[i];
    for(j = N - 1; j > i; j --) {
      X[i] -= X[j] * A[i][j];
    }
    X[i] /= A[i][i];
  }
  return NULL;
}
```

ПАРАЛЛЕЛЬНЫЙ ВАРИАНТ: РАБОЧИЙ ПОТОК

```
void* gauss(void* arg) {
  int myn, j, k, si, cond = 1;
  double maxnorm, e;

  pthread_mutex_lock(&mut);
  myn = tcnt ++;
  pthread_mutex_unlock(&mut);

  while(cond) {
    pthread_mutex_lock(&mut);
    while(dogauss == 0)
      pthread_cond_wait(&cv, &mut);
    pthread_mutex_unlock(&mut);
    for(j = i + 1 + myn; j < N; j += numt) {
      e = A[j][i] / A[i][i];
      B[j] -= e * B[i];
      for(k = i; k < N; k ++) {
        A[j][k] -= e * A[i][k];
      }
    }
```

```
    pthread_mutex_lock(&mute);
      counter ++;
      if(i == (N -2))
        cond = 0;
      if(counter == numt){
        dogauss = 0;
        pthread_cond_broadcast(&cve);
      } else {
        pthread_cond_wait(&cve,
&mute);
      }
      pthread_mutex_unlock(&mute);
    }
    return NULL;
}
```

Зачем нужен while ?

Конструкция

while(GUARD)

pthread_cond_wait

используется для того, чтобы пропускать блокировку при выполненном GUARD и возобновлять ее при срабатывании сигнала, но не выполненном GUARD

Программа ждет выполнения некоторого условия.

Результаты эксперимента (HP Superdome)

numt	(последовательный)	2	4	8	16	32	64
Время (сек)	81	44	24	15	12	15	38

Блокирование на чтение/запись

Несложно заметить, что чтение общих переменных разными потоками не приводит к неоднозначности и/или каким-то конфликтным ситуациям. При доступе к общим переменным нужно запрещать

1) одновременную модификацию данных разными потоками
2) одновременную модификацию и чтение данных

Для этого в PTHREADS предусмотрен специальный вид блокировок – **read-write locks**

Блокирование на чтение-запись
Инициализация и уничтожение

pthread_rwlock_t – тип для блокировки чтения-записи

Инициализация
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
const pthread_rwlockattr_t *restrict attr);

pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;

(эквивалентно attr = NULL)

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

освобождение ресурсов, занятых данной блокировкой

Блокирование на чтение-запись
Чтение

Захват блокировки на чтение

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
блокирует вызывающий поток, если есть еще потоки, захватившие блокировку на запись

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
не блокирует, если не может выполнить захват – возвращает ошибку

Снятие блокировки
int pthread_rwlock_unlock(pthread_rwlock_t *_rwlock_);

Блокирование на чтение-запись

Захват блокировки на запись

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock); - блокирует вызывающий поток, если есть другие потоки захватившие объект

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock); не блокирует вызывающий поток

Пример: параллельная работа со списком данных

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

pthread_rwlock_t rw;


void traverse_list() {
 /** read but not modify the list
**/
 sleep(2);
 std::cout << "traversing ...\n";
 std::cout.flush();
}
```

```
void update_list() {
 /** modify the list **/
 sleep(8);
 std::cout << "updating ...\n";
 std::cout.flush();
}
```

```
void* reader(void* arg) {
  pthread_rwlock_rdlock(&rw);
  traverse_list();
  pthread_rwlock_unlock(&rw);
  return NULL;
}

void* writer(void* arg) {
  pthread_rwlock_wrlock(&rw);
  update_list();
  pthread_rwlock_unlock(&rw);
  return NULL;
}
```

```
#define NT 16

int main(int argc, char* argv[]) {
  pthread_t wrthread1, wrthread2;
  pthread_t threads[NT];
  pthread_rwlock_init(&rw, NULL);
  pthread_create(&wrthread1, NULL, writer, NULL);
  pthread_create(&wrthread2, NULL, writer, NULL);
  for(int i = 0; i < NT; i ++)
    pthread_create(threads + i, NULL, reader, NULL);
  for(int i = 0; i < NT; i ++)
    pthread_join(threads[i], NULL);
  pthread_join(wrthread2, NULL);
  pthread_join(wrthread1, NULL);
  pthread_rwlock_destroy(&rw);
}
```

pthread_once

Иногда требуется проконтролировать, что некоторый блок кода исполняется только один раз в программе. Такой механизм часто применяют для динамической инициализации различных библиотек.

В этом случае можно применить вызов pthread_once.

pthread_once

Инициализация

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

Вызов

```
int pthread_once(pthread_once_t *once_control,
        void (*init_routine)(void));
```

Функция init_routine будет вызываться один лишь раз с помощью механизма pthread_once. Последующие обращения не приведут к вызову функции.

pthread_once

- Гарантируется, что функция init_routine, вызывается не более одного раза
- Гарантируется, что после завершения вызова pthread_once в некотором функция init_routine была выполнена хотя бы один раз, и потом может безопасно пользоваться фактом инициализации

Данные, ассоциированные с потоком

Иногда требуется связать с потоком какие-то данные, например указатель на область памяти. Причем эти данные должны быть доступны из любой функции, выполняемой в данном потоке.

Сложность состоит в том, что в многопоточной среде эти данные не могут храниться в глобальных переменных и их придется тогда передавать во все функции программы в качестве аргумента, что крайне затруднительно.

Данные, ассоциированные с потоком

```
int pthread_key_create(pthread_key_t *key, void (*destructor)(void*));
```

- Создает ключ key для всех потоков
- С одним и тем же ключом в разных потоках связываются разные данные
- Деструктор вызывается при завершении каждого потока, если значение, связанное с ключом не нулевое и при создании ключа указатель на деструктор был ненулевой. В качестве аргумента деструктор получает ассоциированное с ключом значение.

Связывание ключа и значения

int pthread_setspecific(pthread_key_t key, const void *value);

Связывает с ключом key указатель value.

void *pthread_getspecific(pthread_key_t key);

Возвращает указатель, связанный с ключом key. Если ничего не связано, то возвращается NULL.

Удаление ключа

```
int pthread_key_delete(pthread_key_t key);
```

Удаляет ключ, передаваемый в качестве параметра.

```
static pthread_key_t   key;
static pthread_once_t   key_once = PTHREAD_ONCE_INIT;


static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}

void func()
{
    void *ptr;

    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

Обработка завершения потока

void pthread_cleanup_push(void (*routine)(void*), void *arg);

Помещает в стек обработчик завершения routine, связанный с аргументом arg.

void pthread_cleanup_pop(int execute);

Извлекает обработчик завершения потока из стека и (если execute != 0), выполняет его с аргументом arg.

Обработчики завершения потока вызываются при его завершении в порядке, обратном помещению в стек.

Остановка потока

```
int pthread_cancel(pthread_t thread);
```

- Приводит к завершению потока. При этом освобождаются все тред-специфичные данные (связанные с ключами), выполняются все обработчики завершения.
- Возвращаемым значением завершенного потока будет PTHREAD_CANCELED
- Может быть полезно для освобождения ресурсов, когда в завершении выполняемой потоками работы нет необходимости (например найдены нужные данные в массиве).

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

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int tn = 0;

void* f(void* p) {
  int myn;
  pthread_mutex_lock(&m);
  myn = tn ++;
  pthread_mutex_unlock(&m);
  for(int i = 0; i < 10; i ++) {
    sleep(1);
    std::cout << "[" << myn << "]\n";
    std::cout.flush();
  }
}
```

```
int main()
{

  int nt = 8;
  pthread_t thr[nt];
  for(int i = 0; i < nt; i ++)
    pthread_create(thr + i, NULL, f, NULL);
  sleep(3);
  for(int i = 0; i < nt; i += 2)
    pthread_cancel(thr[i]);
  for(int i = 0; i < nt; i ++){
    void* arg;
    pthread_join(thr[i], &arg);
    std::cout << "Thread " << i << " was ";
    if(arg == PTHREAD_CANCELED)
      std::cout << "cancelled\n";
    else
      std::cout << "terminated normally\n";
  }
}
```


volatile

static **volatile** int foo;

Ключевое слово volatile (спецификатор типа данных) сообщает, что любая операция доступа к переменной с данным спецификатором (не важно, чтение или записи) имеет побочный эффект и не может быть проигнорирована при оптимизации.

Применяется для работы с внешними устройствами через адреса памяти.

Тонкости применения volatile

```
static int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

```
void bar_optimized(void) {
    foo = 0;

    while (true)
        ;
}
```

Тонкости применения volatile

```
static volatile int foo;

void bar(void) {
    foo = 0;

    while (foo != 255)
        ;
}
```

Компилятор рассматривает условие в цикле как операцию с побочным эффектом и не выполняет формальную замену условия.

volatile и многопоточность

```
static int foo;

void bar(void) {
   foo = 0;

   while (true) {
      pthread_mutex_lock(&m);
      if(foo == 255) {
        pthread_mutex_unlock(&m);
        break;
      }
      pthread_mutex_unlock(&m);
   }
}
```

Компилятор не выполняет оптимизацию, т.к. присутствует вызов функций, которые могут имеют эффект синхронизации памяти и компилятор не имеет права оптимизировать код, заменяя условие.

volatile не нужен в многопоточном программировании

Thread-safe библиотеки

Для использования в многопоточных приложениях библиотеки должны предоставлять thread-safe вызовы, т.к. вызовы, создающие локальные копии модифицируемых общих данных.

Лабораторные работы

- Реализовать умножение матриц с использованием PTHREADS, измерить ускорение при различном количестве потоков
- Реализовать численное интегрирование с использованием PTHREADS, измерить ускорение при различном числе потоков

Спасибо за внимание!

(Ваши вопросы)