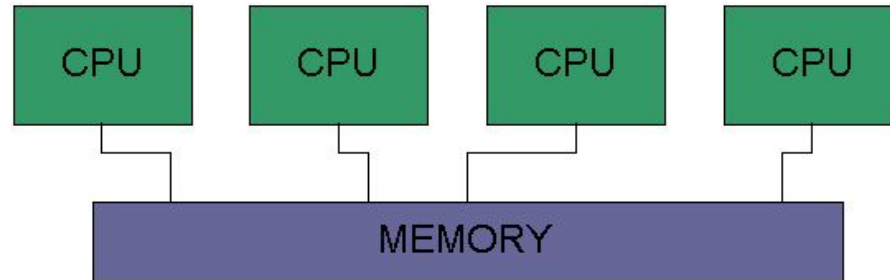


Многопоточное программирование

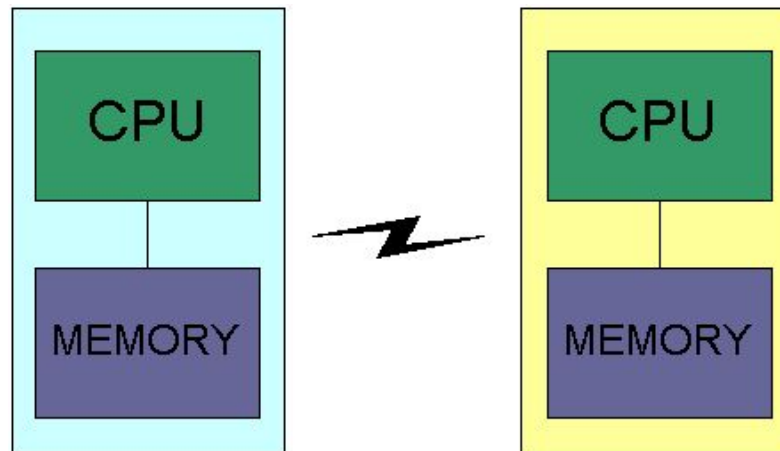


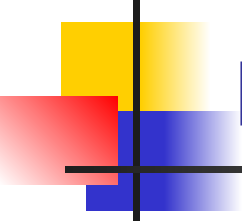
Виды параллелизма.

Общая память



Распределенная
память





Средства параллельного программирования

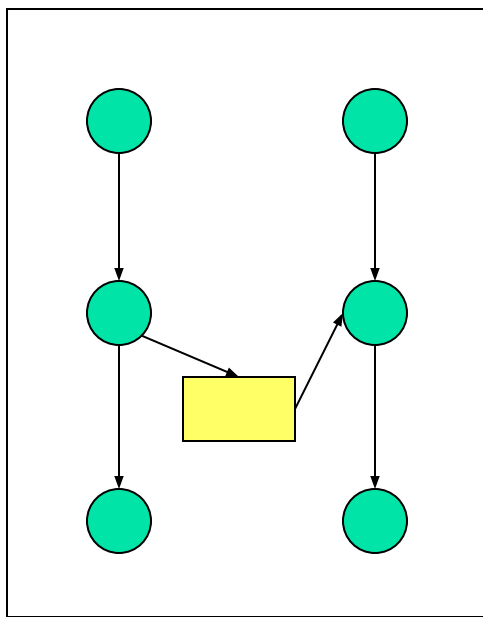
	Общая память	Распределенная память
Системные средства	threads	sockets
Специальные библиотеки	OpenMP	MPI PVM



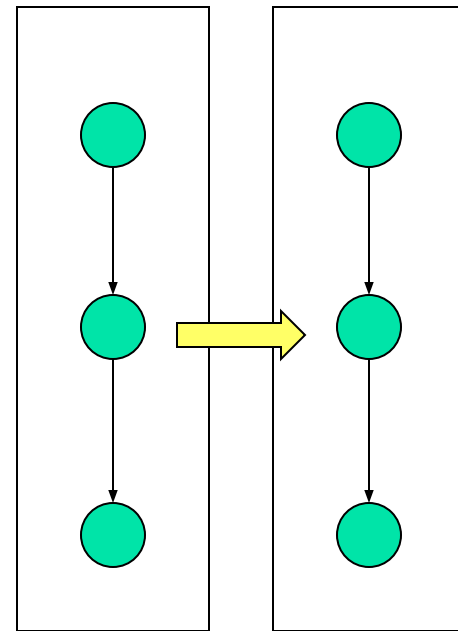
Треды

Тредами (потоки, легковесные процессы) называются параллельно выполняющиеся потоки управления в адресном пространстве одного процесса.

Треды и процессы



обмен
через
посылку
сообщений



обмен через общую
память



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».



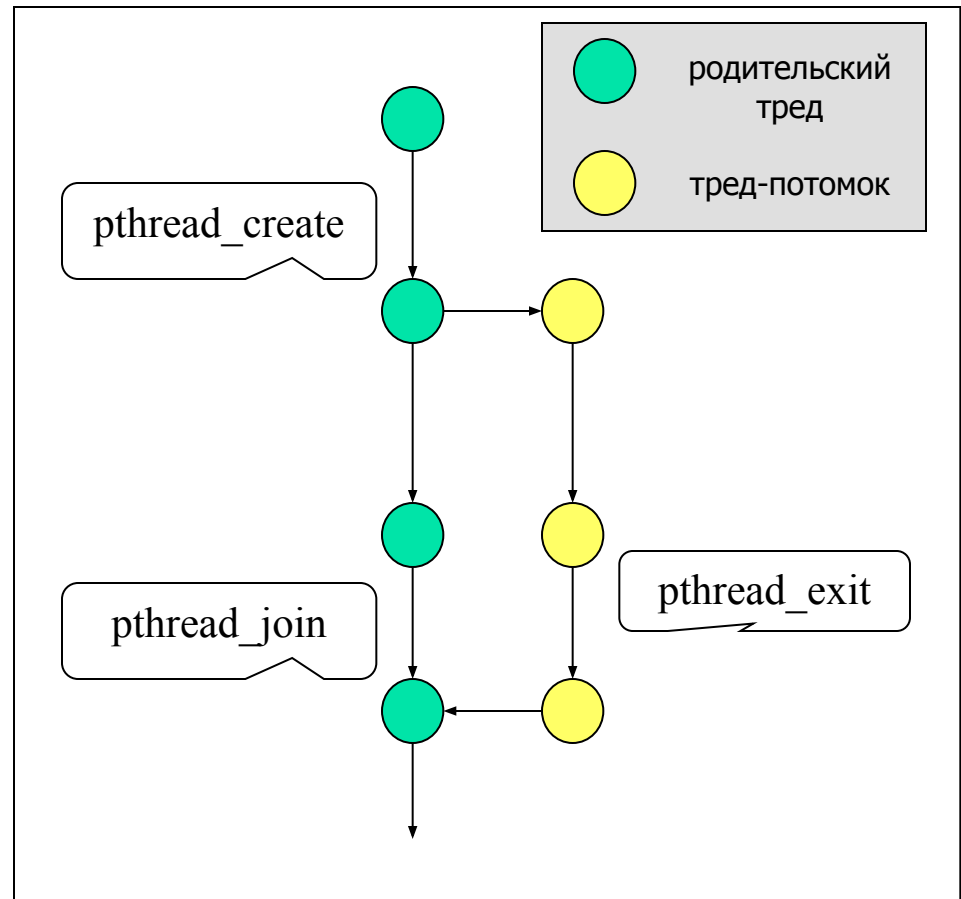
Средства многопоточного программирования

Треды поддерживаются практически всеми современными операционными системами. Средства для многопоточного программирования встроены в язык Java.

Переносимая библиотека [pthreads](#), разработанная Xavier Leroy, предоставляет средства для создания и управления тредами.

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;

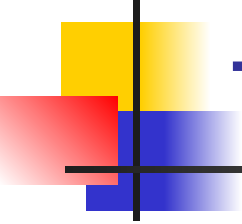


Завершение треда

```
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

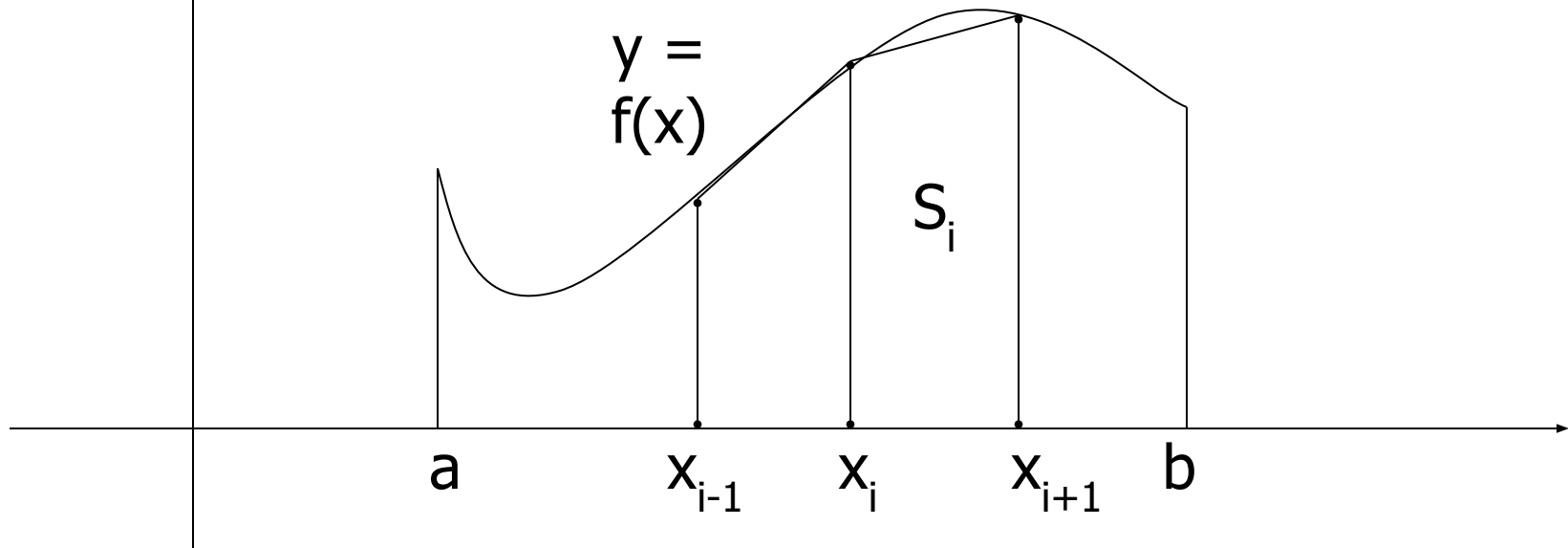
```
int pthread_join( pthread_t inHandle, void **outReturnValue);
```

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

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

Пример: вычисление определенного интеграла

$$\int_a^b f(x) dx \approx \sum S_i$$
$$S_i \approx h \cdot (f(x_{i+1}) + f(x_i)) / 2$$



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

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

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

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

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

    my = *(int*)addr;

    s = 0.0;
    for(p = a + my * h; p < b; p += numt * h)
    {
        s += h*(f(p) + f(p + h))/2.;
    }
    r[my] = s;
    return (void*)(r + my);
}
```

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

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

    for(i = 0; i < numt; i++)
        pthread_create(threads + i, NULL, worker, (void*)&i);

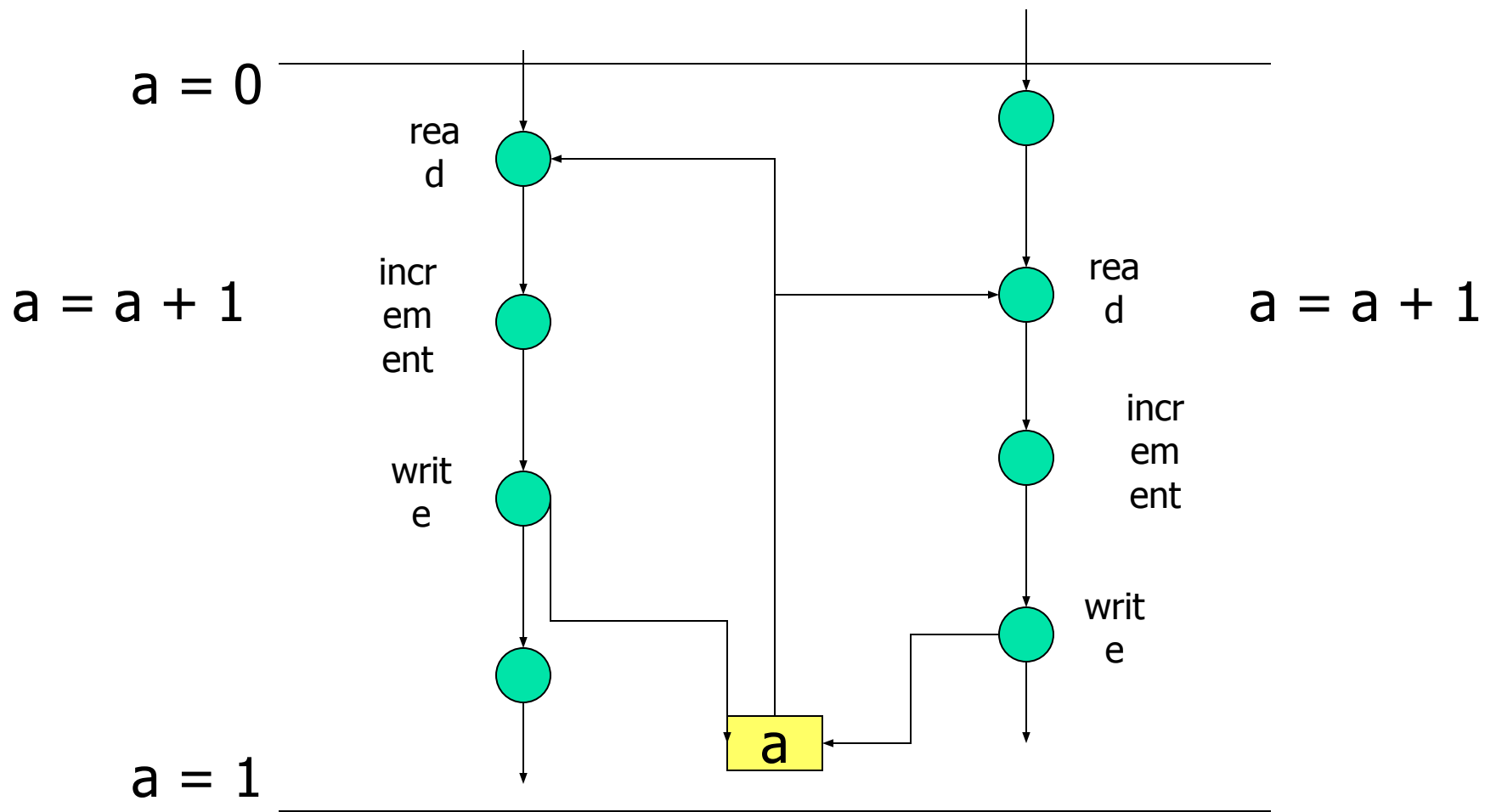
    S = 0;
    for(i = 0; i < numt; i++) {
        double* r;

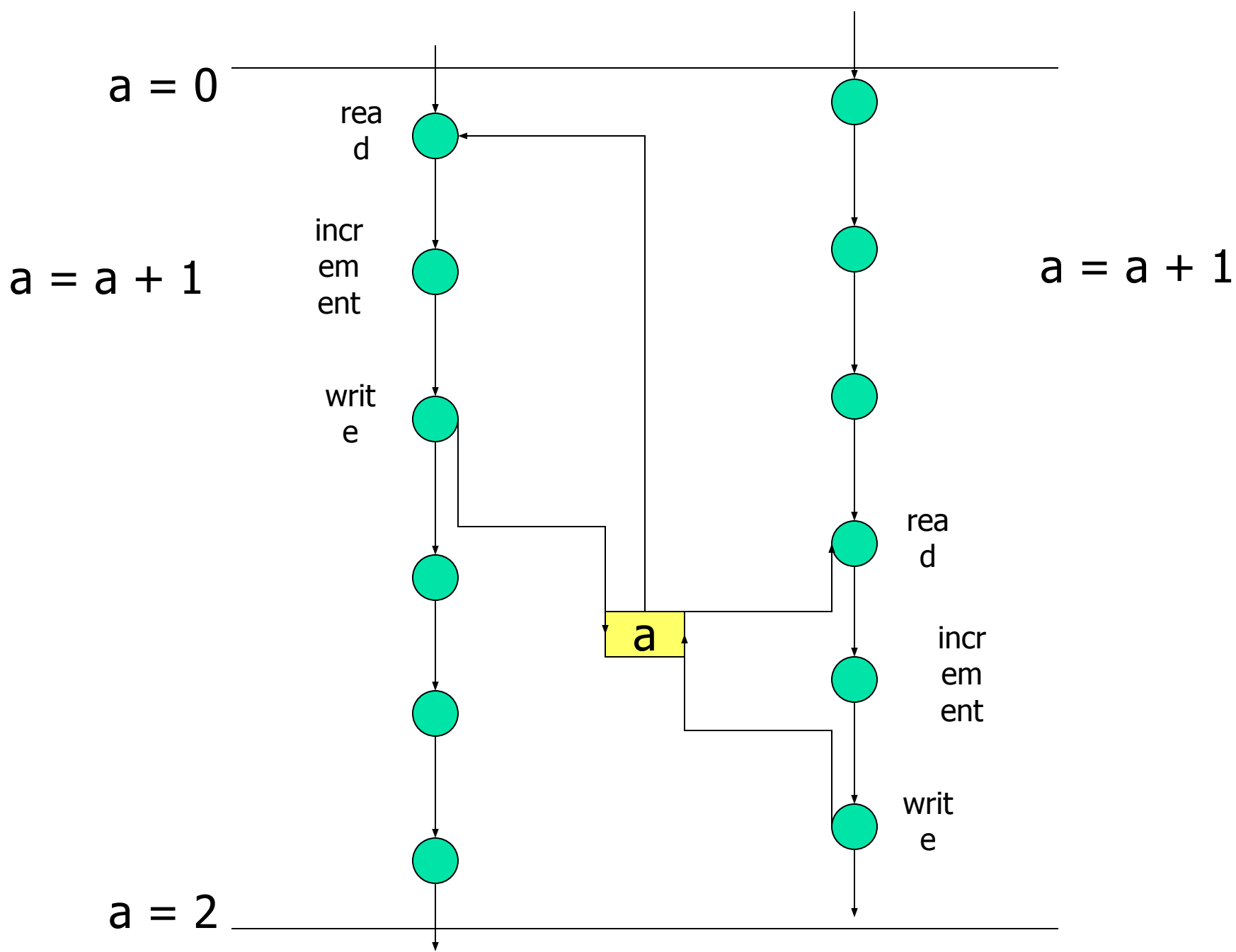
        pthread_join(threads[i], (void**)&r);
        S += *r;
    }
    printf("pi = %lf\n", S);
}
```



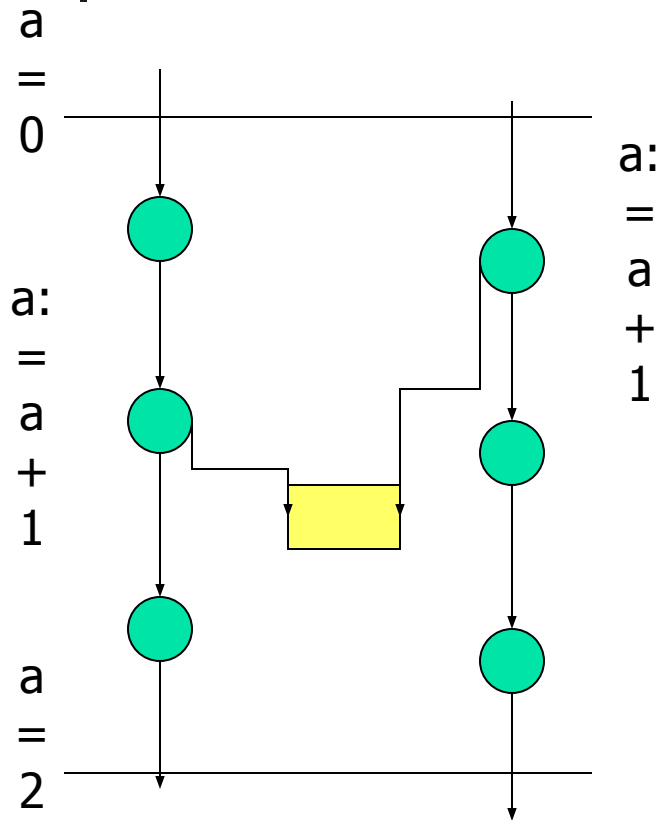
Проблема недетерминизма

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





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 значения 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)

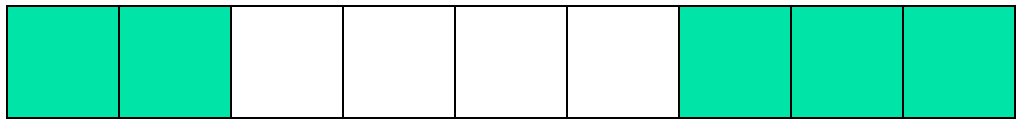
value – начальное значение

sem_post(sem_t* semaphor) – увеличение семафора

sem_wait(sem_t* semaphor) – уменьшение семафора



Кольцевой буфер



↑
front

↑
rear

producer

consumer

```
#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);
    }
}
```

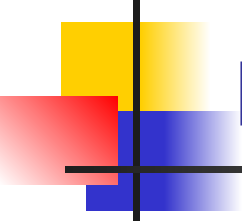
```
main (int argc, char *argv[])
{
    pthread_t pt;
    pthread_t ct;

    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 выполняться только одним тредом.



Пример: создание неделимой операции

1-й тред:

$a = a + 1$

2-й тред:

$a = a + 1$

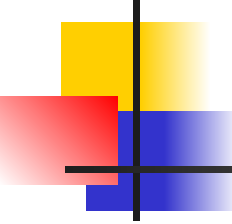
Как сделать операцию $a = a + 1$
неделимой?

1-й тред:

```
while(true) {  
    while(in2);  
    in1 = true;  
    a = a + 1;  
    in1 = false;  
}
```

2-й тред:

```
while(true) {  
    while(in1);  
    in2 = true;  
    a = a + 1;  
    in2 = false;  
}
```



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);

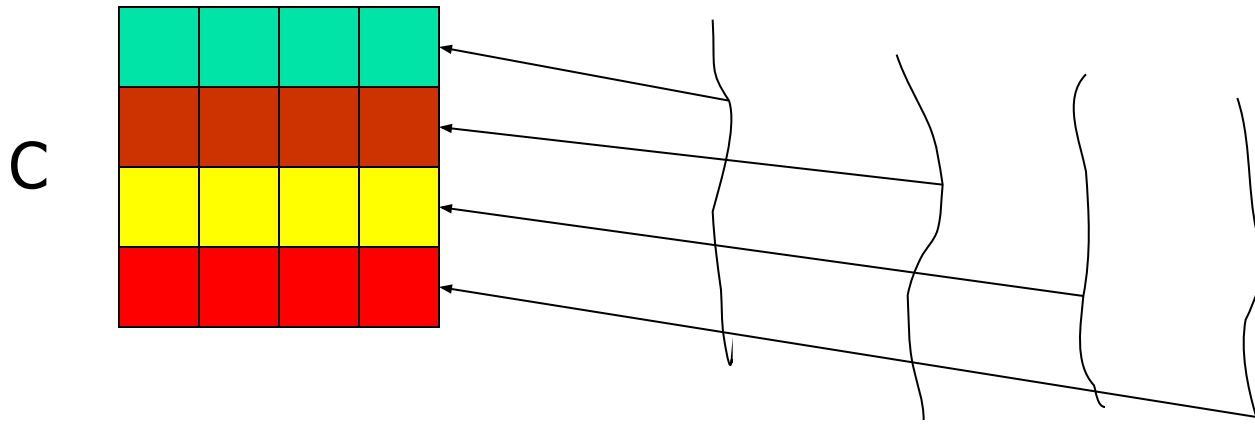
Захват и освобождение мютекса:

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



$$C = A * B$$

каждый тред вычисляет свою строку матрицы



Умножение матриц: код программы

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

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

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

```
pthread_mutex_t mut;
```

```
static int N, nrow;
```

```
static double *A, *B, *C;
```



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

    A = malloc (N * N * sizeof (double));
    B = malloc (N * N * sizeof (double));
    C = 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, j;  
    while (nrow < N)  
    {  
        int oldrow;  
  
        pthread_mutex_lock (&mut);  
        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;
}
```

```
main (int argc, char *argv[])
{
    int i, nthreads;
    pthread_t *threads;

    pthread_mutex_init(&mut, NULL);

    nthreads = atoi (argv[1]);
    threads = malloc (nthreads * sizeof (pthread_t));

    N = atoi (argv[2]);
    setup_matrices ();
```

```
for (i = 0; i < nthreads; i++)  
    pthread_create (threads + i, NULL, worker, NULL);
```

```
for (i = 0; i < nthreads; i++)  
    pthread_join (threads[i], NULL);
```

```
if (argc > 3)  
    print_result ();
```

```
pthread_mutex_destory(&mut);
```

```
}
```