Именованные каналы

- Канал, доступ к которому выполняется через точку привязки файловой системы
- Ядро создает по одному объекту именованного канала для каждой записи в файловой системе

```
int mkfifo(const char *path, mode_t mode);
```

- Создание специального объекта в файловой системе
- Удаление — с помощью unlink

Открытие именованного канала

- Открывается с помощью системного вызова open:

```
fdr = open(path, O_RDONLY, 0); // на чтение
fdw = open(path, O_WRONLY, 0); // на запись
```

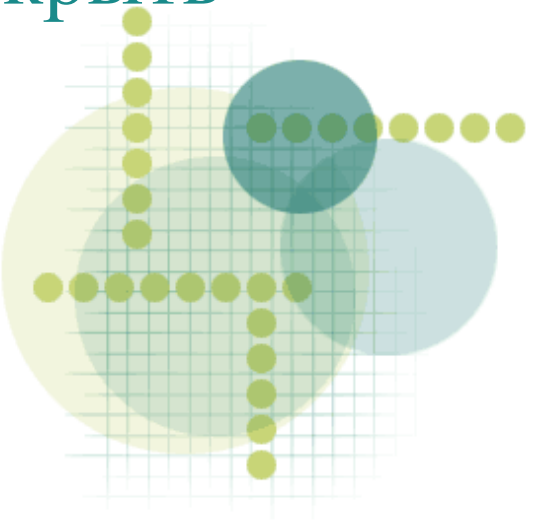
- Операции открытия блокируются до выполнения противоположной операции
  - Открытие на чтение заблокирует процесс, пока другой процесс не откроет на запись
  - Открытие на запись заблокирует процесс, пока другой процесс не откроет на чтение
- После открытия работа — как с обычным каналом

Открытие канала

- Допускается открытие именованного канала в режиме O_RDWR, тогда канал откроется независимо от наличия читателей.

```
fdw = open(path, O_RDWR, 0);
fdr = open(path, O_RDONLY, 0);
```

- Таким образом можно полностью открыть канал в одном процессе



Нити

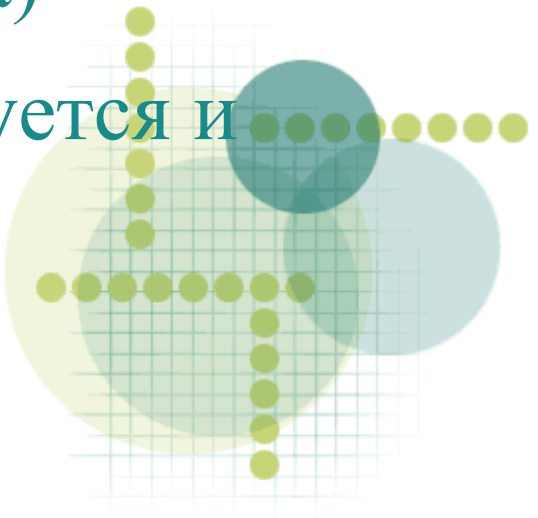
- Нить (легковесный процесс) — единица планирования времени ЦП в рамках одного процесса
- Все нити разделяют общее адресное пространство, открытые файловые дескрипторы и т. д.
- Каждая нить имеет свой стек, свой набор регистров

Главная нить

- При создании процесса создается главная (основная) нить. В рамках основной нити вызывается функция main.
- Процесс завершается, когда завершаются все нити процесса

Уровни планирования нитей

- 1:1 — каждая нить процесса планируется к выполнению на уровне ядра (ядро хранит информацию о всех нитях процесса)
- N:1 — все нити процесса планируются к выполнению на уровне процесса (ядро не имеет информации о нитях процесса)
- N:M — гибридный подход, используется и планирование на уровне процесса, и планирование на уровне ядра



Уровни планирования нитей

- Планирование на уровне процесса — меньше накладные расходы (нет переключения в режим ядра), но необходимо специальным образом работать с системными вызовами, которые могут заблокировать процесс, так как будут заблокированы все нити
- Планирование на уровне ядра — возрастает нагрузка на планировщик процессов, но отсутствуют проблемы с блокирующими вызовами. Кроме того, ядро может запускать нити параллельно на нескольких ядрах или процессорах.

POSIX threads (pthread)

- Стандарт на интерфейс работы с нитями
- Реализован во всех UNIX-системах
- Прототипы функций и типов данных объявлены в файле `<pthread.h>`
- Функции и типы данных имеют префикс `pthread_`
- Для компиляции программы с помощью gcc используется ключ -pthread

```
gcc -Wall -g -pthread prog.c -oprog
```

Главная функция нити

- Главная функция запускается при старте нити
- Когда главная функция завершает работу, нить уничтожается

```
void *start_routine(void *arg);
```

- Параметр arg позволяет передавать любой указатель в нить
- Возвращаемое значение передается в нить, которая ожидает завершения данной нити

Создание нити

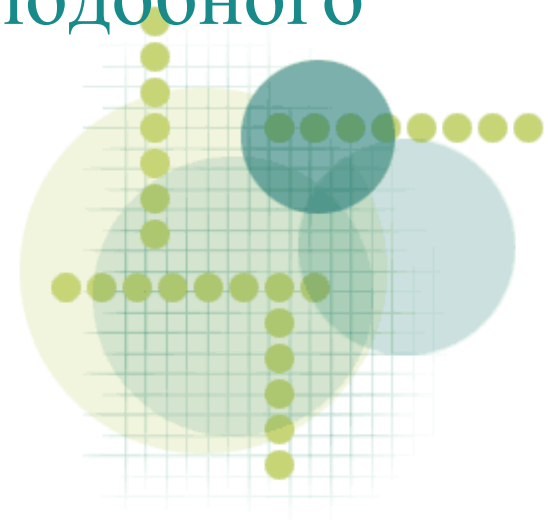
```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *arg);
```

- thread — переменная для сохранения идентификатора нити
- attr — дополнительные атрибуты создаваемой нити
- start_routine — главная функция
- arg — параметр, передаваемый в главную функцию

Завершение нити

```
void pthread_exit(void *value_ptr);
```

- Завершение работы нити, value_ptr — возвращаемое значение
- Если нить выполняет недопустимую операцию (то, что на уровне процесса вызвало бы посылку сигнала SIGSEGV или подобного ядром), завершается весь процесс



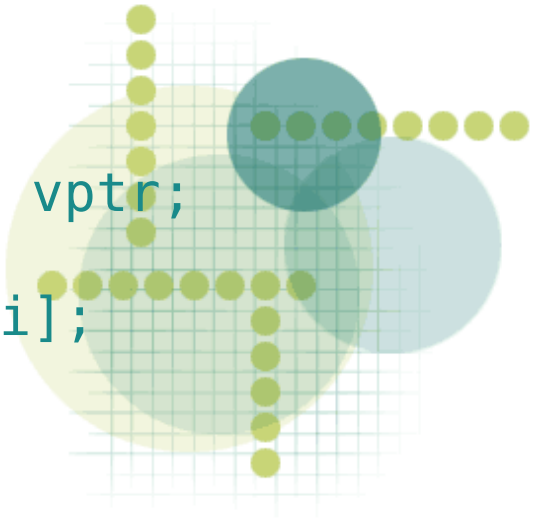
Ожидание завершения нити

```
int pthread_join(pthread_t thr, void **vptr);
```

- thr — идентификатор нити, завершение которой ожидается
- vptr — адрес переменной, в которую будет записан указатель, возвращенный главной функцией нити
- Если нить thr выполняется, текущая нить будет приостановлена до окончания работы нити thr
- Нить можно ждать только один раз
- Для созданной нити либо должна быть выполнена операция join, либо нить должна быть объявлена фоновой

Пример: параллельное суммирование

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
enum { N = 1024 * 1024 };
struct task_data
{
  double *data;
  int n;
  double s;
};
void *thread_func(void *vptr)
{
  struct task_data *pd = (struct task_data*) vptr;
  int i; double s = 0.0;
  for (i = 0; i < pd->n; ++i) s += pd->data[i];
  pd->s = s;
  return pd;
}
```



Параллельное суммирование

```
int main(void)
{
  double *data = calloc(N, sizeof(data[0]));
  int i;
  pthread_t th1, th2;
  struct task_data t1, t2;
  srand(time(0));
  for (i=0; i<N; ++i) data[i]=rand()/(RAND_MAX+1.0);
  t1.data = data; t1.n = N / 2;
  t2.data = data + N / 2; t2.n = N - N / 2;
  pthread_create(&th1, NULL, thread_func, &t1);
  pthread_create(&th2, NULL, thread_func, &t2);
  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  printf("%.10g\n", (t1.s + t2.s) / N);
  return 0;
}
```

Уничтожение нити

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

- Запросить уничтожение нити thread
- Нить может быть уничтожена только в специальных точках выполнения — cancellation points
- Если нить объявлена как асинхронно-прерываемая, на функции, используемые нитью, накладываются ограничения
- См. функции pthread_setcancelstate, pthread_setcanceltype

Параллельные процессы

- Параллельные процессы (нити) — процессы (нити), выполнение которых хотя бы частично перекрывается по времени
- Независимые процессы (нити) — используют независимые ресурсы
- Взаимодействующие процессы (нити) — используют ресурсы совместно, выполнение одного может оказать влияние на результат другого
- **Результат выполнения не должен зависеть от порядка переключения между процессами**

Разделяемые ресурсы

int amount = 100;

```
void retrieve(int m)
{
  amount -= m;
}
```

retrieve(30);

```
movl   amount, %eax
subl   $30, %eax
movl   %eax, amount
```

```
void deposit(int m)
{
  amount += m;
}
```

deposit(10);

```
movl   amount, %eax
addl   $10, %eax
movl   %eax, amount
```

Результат?

Разделяемые ресурсы

int amount = 100;

```
void retrieve(int m)
{
  amount -= m;
}
```

retrieve(30);

```
movl   amount, %eax
subl   $30, %eax
movl   %eax, amount
```

```
void deposit(int m)
{
  amount += m;
}
```

deposit(10);

```
movl   amount, %eax
addl   $10, %eax
movl   %eax, amount
```

Правильный: 80
Неправильный: 70
Неправильный: 110

Гонки (race condition)

- Результат работы зависит от порядка переключения выполнения между параллельными процессами
- Очень сложно обнаруживаемые ошибки
- Могут проявляться очень редко при редкой комбинации условий

Критическая секция

- Взаимное исключение — способ работы с разделяемым ресурсом, при котором во время работы процесса (нити) с разделяемым ресурсом другие процессы (нити) не имеют доступ к разделяемому ресурсу
- Критическая секция — фрагмент кода процесса, который выполняется в режиме взаимного исключения

Требования к механизмам взаимного исключения

- Корректность: только один процесс может находиться в критической секции в каждый момент
- Не должно быть никаких предположений о количестве процессоров или скорости работы процессов
- Процесс вне критической секции не должен быть причиной блокировки других процессов
- Справедливость: не должна возникать ситуация, когда некоторый процесс никогда не получит доступа в критическую секцию
- Масштабируемость: процесс в состоянии ожидания не должен расходовать процессорного времени

Пример (наивный)

```
int s = 1;
int amount = 100;
```

```
void lock()
{
  while (s == 0) ;
  s = 0;
}
void unlock()
{
  s = 1;
}
```

Не обеспечивается корректность!
Используется активное ожидание!

Требуется: атомарность операции проверки значения и установки его в 0, изменение состояния ожидающего процесса, оповещение ожидающих процессов

```
void retrieve(int m)
{
  lock();
  amount -= m;
  unlock();
}
```

```
void deposit(int m)
{
  lock();
  amount += m;
  unlock();
}
```

Семафор

- Семафор — это переменная s (целого типа), над которой можно выполнять две операции:
- P(s,v) — если значение s >= v, то s = s - v; в противном случае процесс блокируется — помещается в список процессов, ожидающих освобождения данного семафора
- V(s,v) — s = s + v
- Операции P и V атомарны

Монитор

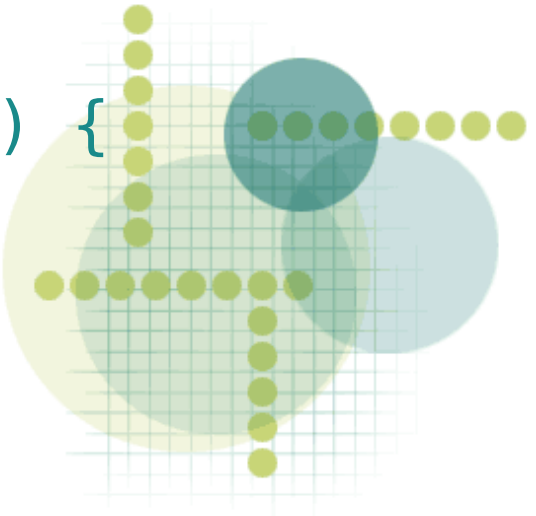
- Монитор — это совокупность некоторых переменных и процедур
- В каждый момент времени может выполняться не более одной процедуры, манипулирующей с этими переменными
- Поддержка мониторов находится на уровне языка программирования (Ada, Java, C#)
- Обычная реализация монитора — с помощью семафоров

Пример монитора

```
class Account
{
  private int amount;
  public synchronized void deposit(int m)
  {
    amount += m;
  }
  public synchronized void retrieve(int m)
  {
    amount -= m;
  }
}
```

Реализация монитора

```
struct Account {
  semaphore lock;
  int amount;
};
void deposit(struct Account *a, int m) {
  P(&a->lock, 1);
  a->amount += m;
  V(&a->lock, 1);
}
void deposit(struct Account *a, int m) {
  P(&a->lock, 1);
  a->amount -= m;
  V(&a->lock, 1);
}
```



Мьютексы

- Мьютекс (mutex — mutual exclusion) — это специальный вид семафора
- Мьютекс может находиться в состоянии 0 (закрыт) и в состоянии 1 (открыт)
- У закрытого мьютекса есть процесс-владелец, только владелец может открыть мьютекс.

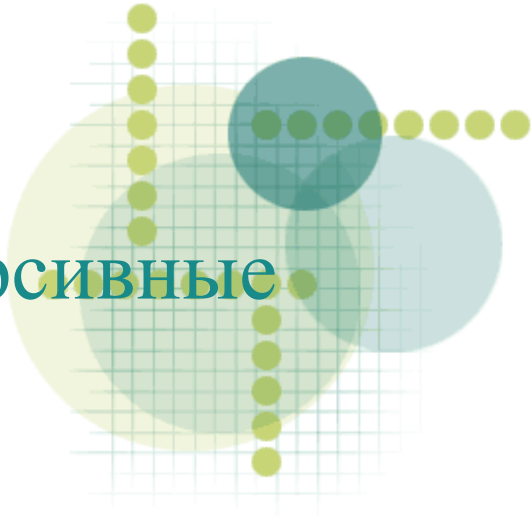
Средства взаимного исключения в pthread

- Мьютексы
- Условные переменные
- Блокировки чтения/записи

Мьютексы pthread

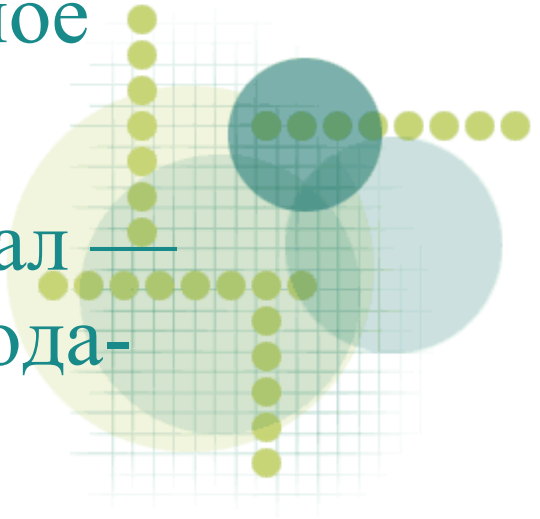
```
int pthread_mutex_init(pthread_mutex_t *mutex,|
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- Мьютекс должен быть предварительно проинициализирован
- Исходное состояние — открыт
- Различаются рекурсивные и нерекурсивные мьютексы



Ожидание наступления события

- Часто требуется, чтобы одна нить ждала наступление некоторого условия
- Например, главная нить может дожидаться завершения расчетов созданных нитей чтобы объединить результаты расчетов нитей
- Вариант решения: мьютекс + активное ожидание — не подходит
- Вариант решения: использовать канал — требует использования операций ввода-вывода



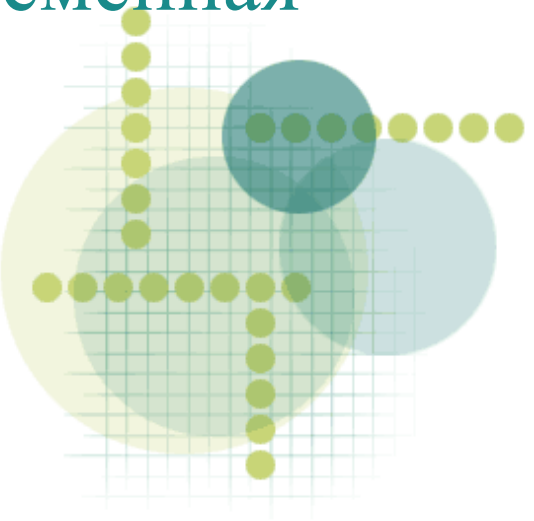
Условные переменные

- Механизм для рассылки уведомлений
- Одна или несколько нитей ждут наступления события (заблокированы)
- При наступлении события нить посылает уведомление ожидающим нитям, пробуждая одну из них или все
- Для блокировки доступа к условной переменной используется мьютекс

Условные переменные pthread

```
int pthread_cond_init(pthread_cond_t *c,
                      const pthread_contattr_t *a);
int pthread_cond_destroy(pthread_cond_t *c);
int pthread_cond_wait(pthread_cond_t *c,
                      pthread_mutex_t *m);
int pthread_cond_broadcast(pthread_cond_t *c);
int pthread_cond_signal(pthread_cond_t *c);
```

- Перед использованием условная переменная должна быть проинициализирована



Пример: ожидание всех рабочих нитей

- Пусть есть N рабочих нитей и есть главная нить, которая ожидает прохождения контрольной точки

```
// условная переменная
pthread_mutex_t wait_mutex;
pthread_cond_t wait_cond;
int wait_count;
// рабочие нити
pthread_mutex_lock(&wait_mutex);
if (++wait_count == N)
  pthread_cond_signal(&wait_cond);
pthread_mutex_unlock(&wait_mutex);
```

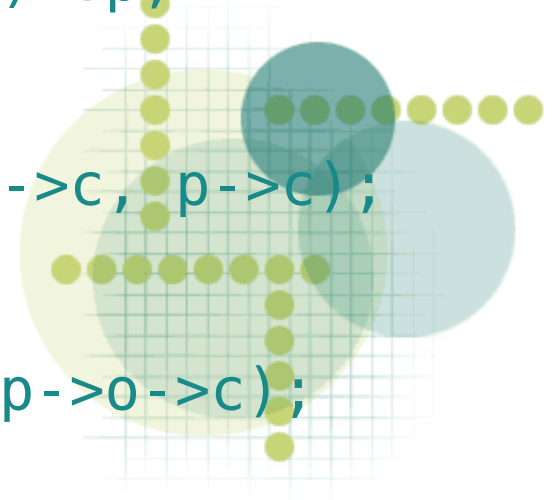
Пример: ожидание всех рабочих нитей

- Пусть есть N рабочих нитей и есть главная нить, которая ожидает прохождения контрольной точки

```
// условная переменная
pthread_mutex_t wait_mutex;
pthread_cond_t wait_cond;
int wait_count;
// главная нить
pthread_mutex_lock(&wait_mutex);
while (wait_count != N)
  pthread_cond_wait(&wait_cond, &wait_mutex);
pthread_mutex_unlock(&wait_mutex);
```

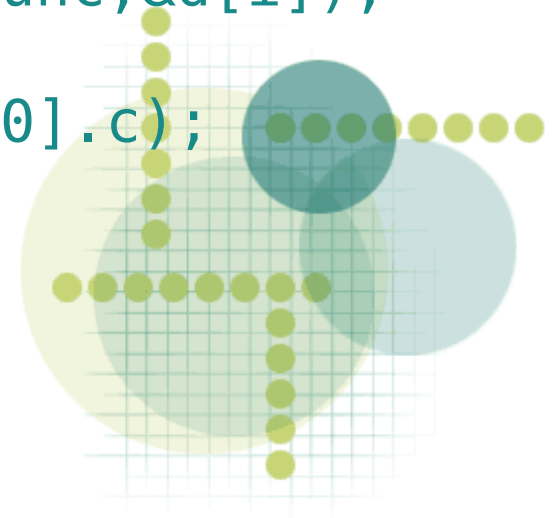
Пример: запуск нитей по очереди (пинг-понг)

```
int count;
struct thr_info {
  int s; pthread_t t; int flag;
  pthread_cond_t c;
  pthread_mutex_t m;
  struct thr_info *o;
};
void *thread_func(void *vp)
{ struct thr_info *p = (struct thr_info*) vp;
  while (1) {
    pthread_mutex_lock(p->m);
    while (!p->flag) pthread_cond_wait(p->c, p->c);
    pthread_mutex_unlock(p->m);
    printf("%d: %d\n", p->s, count++);
    p->o->flag = 1; pthread_cond_signal(p->o->c);
  }
}
```

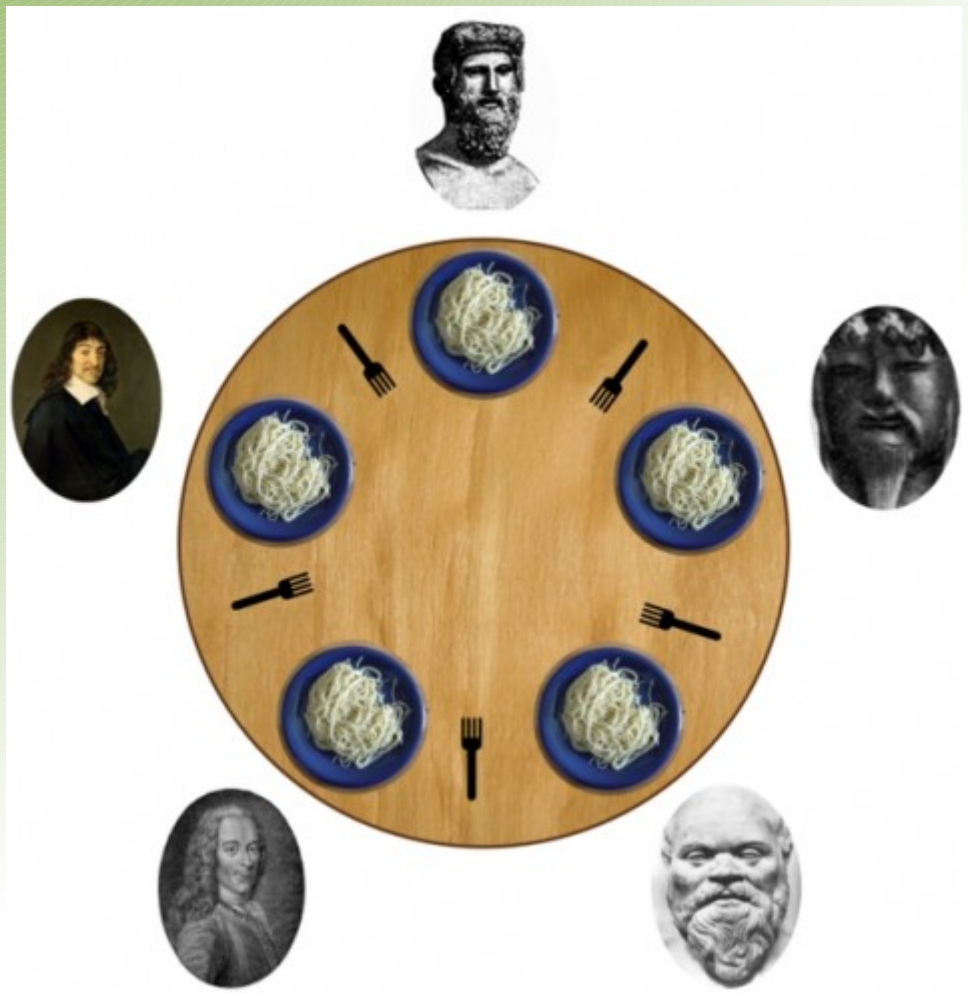


Пример: запуск нитей по очереди (пинг-понг)

```
struct thr_info d[N];
int main(void)
{ int i;
  for (i = 0; i < N; ++i) {
    d[i].s = i + 1;
    pthread_cond_init(&d[i].c, NULL);
    pthread_mutex_init(&d[i].m, NULL);
    d[i].o = &d[(i + 1) % N];
    pthread_create(&d[i].t,NULL,thread_func,&d[i]);
  }
  d[0].flag = 1; pthread_cond_signal(&d[0].c);
  for (i = 0; i < N; ++i)
    pthread_join(&d[i].t);
  return 0;
}
```



Обедающие философы



Наивное решение

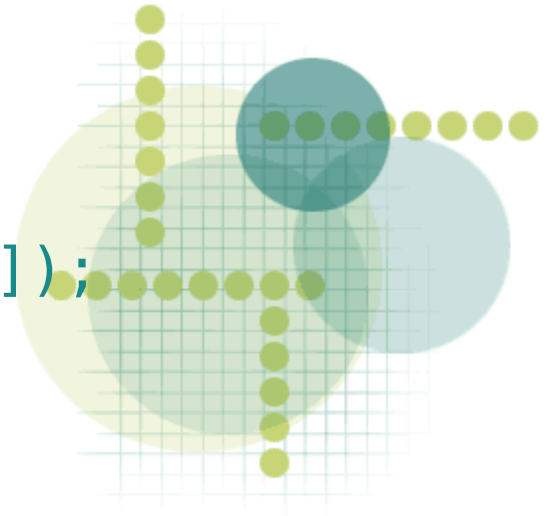
```
void philosopher ( int i ) {
  while (TRUE) {
    think () ;
    take_fork ( i ) ;
    take_fork ( ( i + 1 ) % N ) ;
    eat () ;
    put_fork ( i ) ;
    put_fork ( ( i + 1 )  % N ) ;
  }
  return;

}
```

Deadlock

- Возможна ситуация, когда все философы одновременно захотят есть и возьмут левую от себя вилку — никто не сможет начать есть

Философы, ч. 1

```
enum { THINKING, EATING, HUNGRY };
#define LEFT(i) (((i) + N – 1) % N)
#define RIGHT(i) (((i) + 1) % N)
pthread_mutex_t m;
int state[N];
pthread_cond_t c[N];
void test(int i, int j)
{
  if (state[i] == HUNGRY
      && state[LEFT(i)] != EATING
      && state[RIGHT(i)] != EATING) {
    state[i] = EATING;
    if (i != j) pthread_cond_signal(&c[i]);
  }
}
```



Философы, ч. 2

```
void take_forks(int i) {
  pthread_mutex_lock(&m);
  state[i] = HUNGRY;
  test(i, i);
  while (state[i] == HUNGRY)
    pthread_cond_wait(&c[i], &m);
  pthread_mutex_unlock(&m);
}
void put_forks(int i) {
  pthread_mutex_lock(&m);
  state[i] = THINKING;
  test(LEFT(i), i);
  test(RIGHT(i), i);
  pthread_mutex_unlock(&m);
}
```

Философы, ч. 3

```
void life(int i)
{
  while (1) {
    think();
    take_forks(i);
    eat();
    put_forks(i);
  }
}
```

Блокировка чтения-записи

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwl);
int pthread_rwlock_init(pthread_rwlock_t *rwl,
                const pthread_rwlockattr_t *attr);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwl);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwl);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwl);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwl);
int pthread_rwlock_unlock(pthread_rwlock_t *rwl);
```