

Потоки и синхронизация

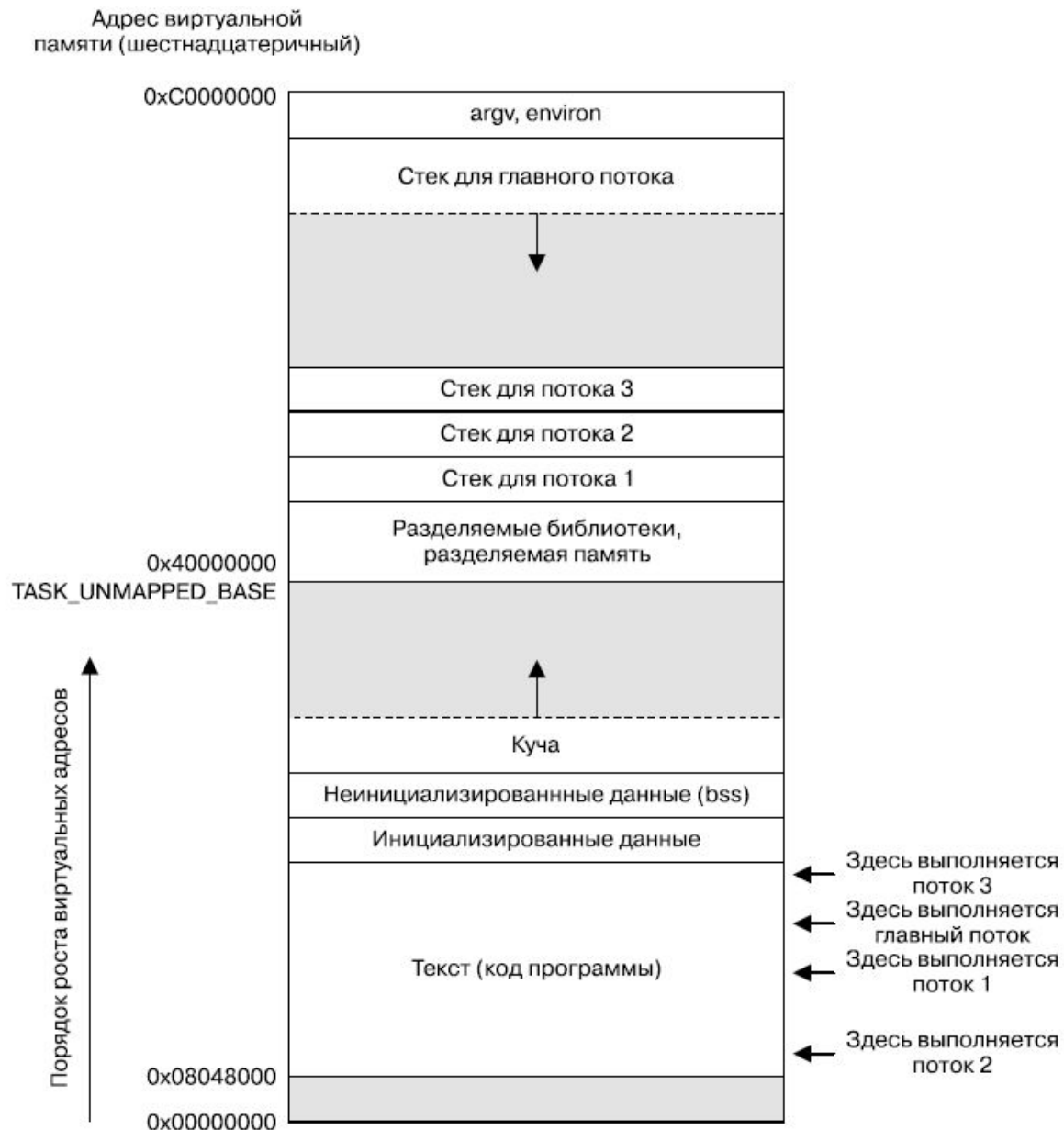


Рис. 29.1. Четыре потока, выполняющиеся внутри процесса (Linux/x86-32)

Потоки vs процессы


- В случае `fork()` родитель и потомок не разделяют память, требуется IPC для обмена данными.
- Создание процесса с помощью `fork()` потребляет относительно много ресурсов. Даже если copy-on-write, приходится дублировать различные атрибуты процесса, такие как таблицы со страницами памяти и файловыми дескрипторами
- Обмен информации между потоками является простым и быстрым. Для этого всего лишь нужно скопировать данные в общие переменные (глобальные или в куче). Но иногда приходится применять методы синхронизации.
- Создание потока занимает меньше времени, чем создание процесса ~ 10 раз. Процедура создания потока является более быстрой, поскольку многие атрибуты вместо непосредственного копирования, как в случае с `fork()`, просто разделяются. В частности, отпадает потребность в дублировании страниц памяти (с помощью копирования при записи) и таблиц со страницами.

Процессы vs потоки

- Работа с сигналами в многопоточном приложении требует тщательного проектирования (как правило, рекомендуется избегать использования сигналов в многопоточных программах).
- В многопоточном приложении все потоки должны выполнять одну и ту же программу (возможно, задействуя разные ее функции). В многопроцессных приложениях разные процессы способны выполнять разные программы.
- Помимо самих данных, потоки разделяют и другую информацию (например, файловые дескрипторы, действия сигналов, текущий каталог, идентификаторы (имена) пользователя и группы). Это может быть как преимуществом, так и недостатком, в зависимости от приложения.

Создание потока

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



`int pthread_join(pthread_t thread, void **retval);` - можно вызывать в любом потоке, поскольку нет иерархии потоков они все равны.

Компилируется и компоуется вместе с **-pthread**

Передача данных потоку и возвращаемое значение

```
struct parameters {
    int count;
};

void * func( void *arg) {
    struct parameters *p = (struct parameters *) arg;
    ...
    ...
    return (void *)(p->count);
}

int main() {
    pthread_t t1;
    void *res;

    struct parameters p;
    p.count = 123;
    pthread_create(&t1, NULL, &func, &p)
    pthread_join(t1, &res)
    printf("return value: %d", (int)res)
}
```

Атрибуты потоков

```
int pthread_attr_init(pthread_attr_t *attr);
```

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

Можно получать и изменять информация об атрибутах нити

Для каждого атрибута существует функция изменяющая его значение

Примеры: `pthread_attr_setdetachstate` ,
 `pthread_attr_setstacksize` ...

`stack_example.c`

Race condition. Борьба за ресурс

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

int a=0;

void * f(void *p){
    int i;
    for(i=0;i<100000;i++)
        a++;
    return p;
}

int main(){
    pthread_t t1,t2;

    pthread_create(&t1,NULL,f,NULL);

    pthread_create(&t2,NULL,f,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    printf("a=%d\n",a);
}
```



Семафоры POSIX

Семафор - объект, ограничивающий количество потоков, которые могут войти в заданный участок кода.

Именованные (для процессов) и неименованные (потоки)

Любой поток может работать с семафором



`sem_wait`
(-1)



`sem_post`
(+1)

E_OVERFLOW

Семафоры POSIX

`sem_init(sem_t *sem, int pshared, unsigned int value);`

`sem_destroy(sem_t *sem);`

`sem_post(sem_t *sem);`

`sem_wait(sem_t *sem);`

`sem_t * sem_open(const char *name, int oflag, ...);`

`sem_close(sem_t *sem);`

***** - `pshared = {0, N}` 0 - потоки, N - процессы

****** - `sem_open` - для именованных семафоров

Mutex (MUTual EXclusion)

Обеспечение атомарного доступа к любым разделяемым ресурсам. Частный случай семафора.

- Всего два состояния (открытое и закрытое)
- Максимум один поток может удерживать мьютекс
- Нельзя закрыть закрытый мьютекс (undefined behavior)
- Только владелец может закрыть мьютекс (отличие от семафора)

DEADLOCK

Who will act first? No one because each of them waits for the other to act.



COP: Thread #1 demands Resource #2 but Criminal owns the LOCK

CRIMINAL: Thread #2 demands Resource #1 but Cop owns the LOCK

CRIMINALS FRIEND: Resource #2, the owner of the LOCK is Cop

HOSTAGE OF CRIMINAL: Resource #1, the owner of the LOCK is CRIMINAL

DEADLOCK (варианты решения проблемы)

1. Введение иерархии мьютексов, захват мьютексов всегда в одном и том же порядке.
2. `int pthread_mutex_trylock(pthread_mutex_t *mutex);` - попытка захвата мьютексов, если хотя бы один захвачен, то освобождаем все и повторяем попытку через некоторое время

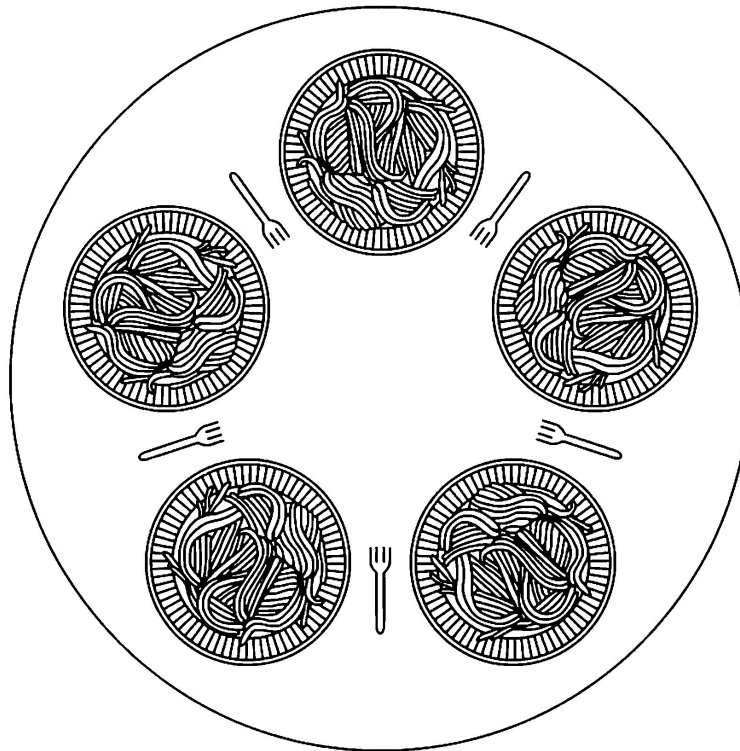
Mutex (MUTual EXclusion)

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

PTHREAD_MUTEX_INITIALIZER - только для статически
размещаемых мьютексов

Задача об обедающих философях

- Каждый философ может либо есть, либо размышлять.
- Философ может есть только тогда, когда держит две вилки – взятую справа и слева
- Каждый может взять ближайшую вилку (если она доступна), или положить – если он уже держит её. Взятие и возвращение каждой вилки являются отдельными действиями, которые должны выполняться одно за другим.



- Ситуация голодной смерти (deadlock)
- Ситуация голодания отдельного философа (resource starvation)

OpenMP

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Компилировать с **-fopenmp**