

**Параллельные вычислительные технологии
Осень 2014 (Parallel Computing Technologies, PCT 14)**

Лекция 4. POSIX Threads.

**Реентерабельность. Сигналы. Локальные
данные. Принудительное завершение.
Шаблоны программирования**

Пазников Алексей Александрович
Кафедра вычислительных систем СибГУТИ

Сайт курса: <http://cpct.sibsutis.ru/~apaznikov/teaching/>

Вопросы: <https://piazza.com/sibsutis.ru/fall2014/pct14/home>

Реентерабельность

Реентерабельность

- Одну функцию может вызывать (re-enter) несколько потоков.
- Функции, которые могут безопасно вызываться из нескольких потоков, называются **безопасными в многопоточной среде или потокобезопасными (thread-safe)**
- Все функции, определённые стандартом Single UNIX Specification, являются потокобезопасными, за исключением ...

Непотребноопасные функции

asctime	ecvt	gethostent	getutxline	putc_unlocked
basename	encrypt	getlogin	gmtime	putchar_unlocked
catgets	endgrent	getnetbyaddr	hcreate	
crypt	endpwent	getnetbyname	hdestroy	putenv
ctime	endutxent	getnetent	hsearch	pututxline
dbm_clearerr	fcvt	getopt	inet_ntoa	rand
dbm_close	ftw	getprotobyname	164a	readdir
dbm_delete	gcvt	getprotobynumb	lgamma	setenv
dbm_error	getc_unlocked	er	lgammaf	setgrent
dbm_fetch	getchar_unlocked	getprotoent	lgammal	setkey
dbm_firstkey	ed	getpwent	localeconv	setpwent
dbm_nextkey	getdate	getpwnam	localtime	setutxent
dbm_open	getenv	getpwuid	lrand48	strerror
dbm_store	getgrent	getservbyname	mrnd48	strtok
dirname	getgrgid	getservbyport	nftw	ttyname
derror	getgrnam	getservent	nl_langinfo	unsetenv
drand48	gethostbyaddr	getutxent	ptsname	wcstombs
	gethostbyname	getutxid		wctomb

Реентерабельность

- У некоторых функций есть альтернативные потокобезопасные версии
- Например, изменяется интерфейс: результат не возвращается в буфере, размещенном статически

acstime__r
ctime_r
getgrgid_r
getgrnam_r
getlogin_r
getpwnam_r
getpwuid_r

gmtime_r
localtime_r
rand_r
readdir_r
strerror_r
strtok_r
ttyname_r

Безопасный способ управления объектами FILE

```
void flockfile(FILE *filehandle);  
int ftrylockfile(FILE *filehandle);  
void funlockfile(FILE *filehandle);
```

Посимвольный ввод-вывод без блокировки:

```
int getc_unlocked(FILE *stream);  
int getchar_unlocked(void);  
int putc_unlocked(int c, FILE *stream);  
int putchar_unlocked(int c);
```

Функции должны быть окружены вызовами
flockfile и funlockfile

Нереентерабельная версия функции getenv

```
static char envbuf[ARG_MAX];
extern char **environ;

char *getenv(const char *name)
{
    int i, len;
    len = strlen(name);

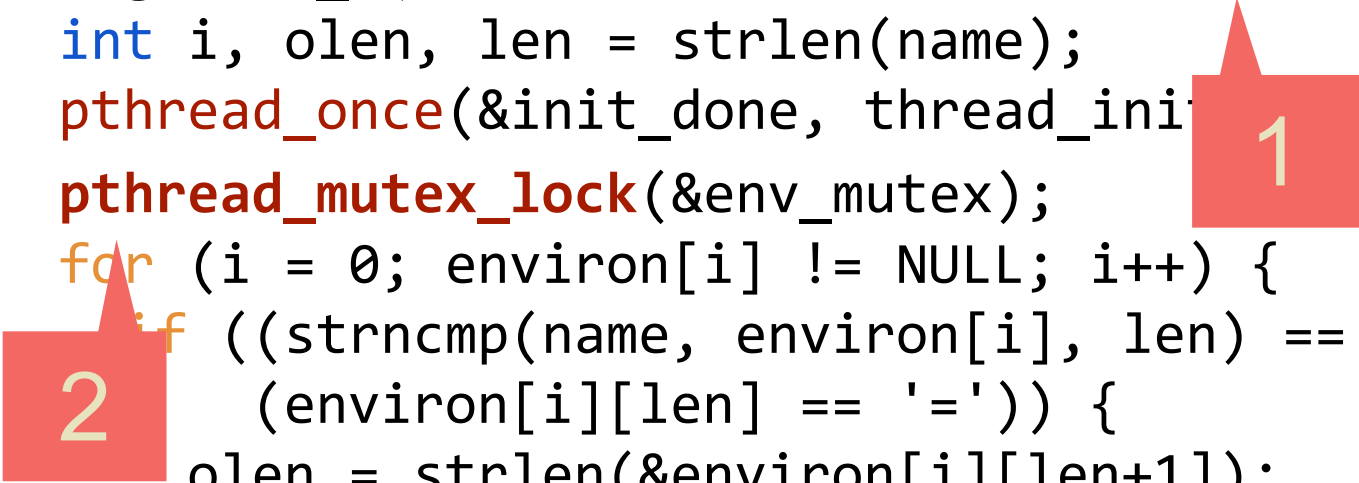
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0)
            && (environ[i][len] == '=')) {
            strcpy(envbuf, &environ[i][len+1]);
            return envbuf;
        }
    }
    return NULL;
}
```

Потокобезопасная версия функции getenv

```
int getenv_r(const char *name, char *buf, int buflen) {
    int i, olen, len = strlen(name);
    pthread_once(&init_done, thread_init);
    pthread_mutex_lock(&env_mutex);
    for (i = 0; environ[i] != NULL; i++) {
        if ((strncmp(name, environ[i], len) == 0) &&
            (environ[i][len] == '=')) {
            olen = strlen(&environ[i][len+1]);
            if (olen >= buflen) {
                pthread_mutex_unlock(&env_mutex); return ENOSPC;
            }
            strcpy(buf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex); return 0;
        }
    }
    pthread_mutex_unlock(&env_mutex); return ENOENT;
}
```


Потокобезопасная версия функции getenv

```
int getenv_r(const char *name, char *buf, int buflen) {  
    int i, olen, len = strlen(name);  
    pthread_once(&init_done, thread_init);  
    pthread_mutex_lock(&env_mutex);  
    for (i = 0; environ[i] != NULL; i++) {  
        if ((strncmp(name, environ[i], len) == 0) &&  
            (environ[i][len] == '=')) {  
            olen = strlen(&environ[i][len+1]);  
            if (olen >= buflen) {  
                pthread_mutex_unlock(&env_mutex); return ENOSPC;  
            }  
            strcpy(buf, &environ[i][len+1]);  
            pthread_mutex_unlock(&env_mutex); return 0;  
        }  
    }  
    pthread_mutex_unlock(&env_mutex); return ENOENT;  
}
```



Потокобезопасная версия функции getenv

```
int getenv_r(const char *name, char *buf, int buflen) {
    int i, olen, len = strlen(name);
    pthread_once(&init_done, thread_init);
    pthread_mutex_lock(&env_mutex);
    for (i = 0; i < N; i++) {
        if ((strcmp(name, environ[i], len) == 0) &&
            (environ[i][len] == '=')) {
            olen = strlen(&environ[i][len+1]);
            if (olen >= buflen) {
                pthread_mutex_unlock(&env_mutex); return ENOSPC;
            }
            strcpy(buf, &environ[i][len+1]);
            pthread_mutex_unlock(&env_mutex); return 0;
        }
    }
    pthread_mutex_unlock(&env_mutex); return ENOENT;
}
```

R/W-мьютексы?

Потокобезопасная версия функции getenv

```
extern char *environ;
pthread_mutex_t envjmutex;

static pthread_once_t initjtone =
    PTHREAD_ONCE_INIT;
static void thread_init(void)
{
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr,
        PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&env_mutex, &attr);
    pthread_mutexattr_destroy(&attr);
}
```

Потокобезопасная версия функции getenv

```
extern char *environ;  
pthread_mutex_t envjmutex;
```

```
static pthread_once_t i
```

```
static void thread_init  
{
```

```
    pthread_mutexattr_t a
```

```
    pthread_mutexattr_ini
```

```
    pthread_mutexattr_settype(&attr,
```

```
                                PTHREAD_MUTEX_RECURSIVE);
```

```
    pthread_mutex_init(&env_mutex, &attr);
```

```
    pthread_mutexattr_destroy(&attr);
```

```
}
```

Зачем нужен
рекурсивный
мьютекс?

Локальные данные ПОТОКОВ

Локальные данные потоков

- Необходимость сохранять данные, **специфичные для конкретного потока.**
- Данные одного потока **не должны быть изменены другим потоком.**
- “Таблица” данных не годится: идентификаторы потоков не целые, не известно заранее, сколько будет потоков.
- Необходим способ для адаптации к многопоточной среде (например, так было в случае с errno)
- Каждый поток может использовать локальные данные, не прибегая к синхронизации.

Локальные данные потоков

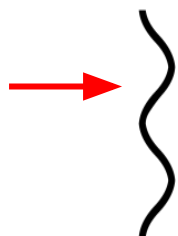
Keys

foo	0
bar	1
buz	2

Destructor

0	fun0()
1	NULL
2	fun2()

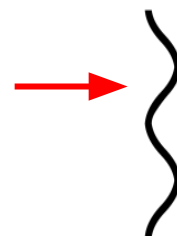
T1



0	x0f98a
1	...
2	...



T2



0	NULL
1	...
2	...

Локальные данные потоков

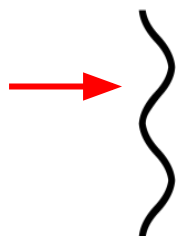
Keys

foo	0
bar	1
buz	2

Destructor

0	fun0()
1	NULL
2	fun2()

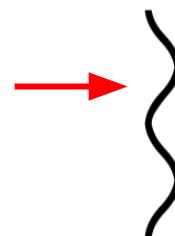
T1



0	x0f98a
1	...
2	...



T2



0	x10df3
1	...
2	...



Создание ключа локальных данных

```
/* Создать ключ, ассоциированный с лок. данными */  
int pthread_key_create(pthread_key_t *key,  
                        void (*destructor)(void*));
```

key - ключ (идентификатор локальных данных)

destructor - деструктор локальных данных

- Ключ может использоваться разными потоками.
- Каждый поток ассоциирует с ключом отдельный набор локальных данных.
- С ключом можно связать функцию-деструктор, которая вызывается в случае штатного (не `exit`, `abort`) завершения потока.
- Поток может создать несколько ключей. По завершении потока запускаются деструкторы для всех ненулевых ключей.

Создание и удаление ключа локальных данных

```
/* Создать ключ, ассоциированный с лок. данными */  
int pthread_key_create(pthread_key_t *key,  
                        void (*destructor)(void*));
```

key - ключ (идентификатор локальных данных)

destructor - деструктор локальных данных

```
/* Разорвать связь ключа с локальными данными */  
int pthread_key_delete(pthread_key_t *key);
```

- Вызов не приводит к деструктору ключа.

Создание ключа локальных данных

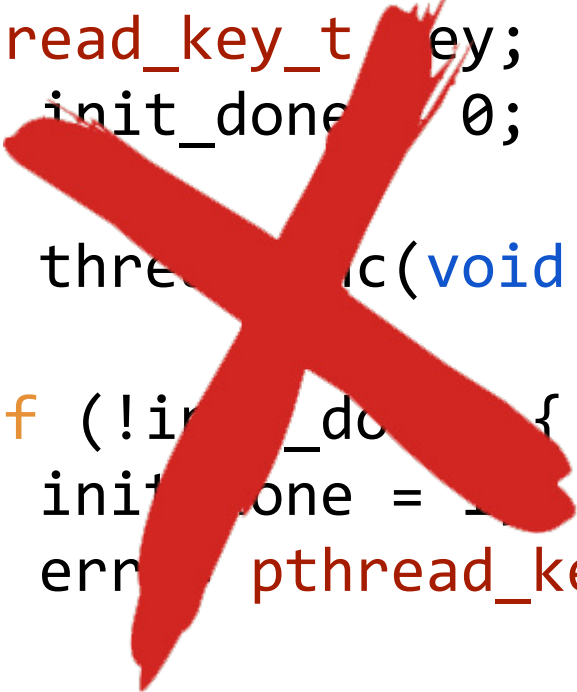
Ключ должен инициироваться атомарно и один раз!

```
void destructor(void *);  
pthread_key_t key;  
int init_done = 0;  
  
int threadfunc(void *arg)  
{  
    if (!init_done) {  
        init.done = 1;  
        err = pthread_key_create(&key, destructor);  
    }  
}
```

Создание ключа локальных данных

Ключ должен инициироваться атомарно и один раз!

```
void destructor(void *);  
pthread_key_t key;  
int init_done = 0;  
  
int thread_func(void *arg)  
{  
    if (!init_done) {  
        init_done = 1;  
        err = pthread_key_create(&key, destructor);  
    }  
}
```



Создание и удаление ключа локальных данных

```
/* функция initfn вызывается один раз при первом  
 * обращении к pthread_once */
```

```
pthread_once_t initflag = PTHREAD_ONCE_INIT;  
int pthread_once(pthread_once_t *initflag,  
                 void (*initfn)(void));
```

```
void destructor(void *);  
pthread_key_t key;  
pthread_once_t init_done = PTHREAD_ONCEINIT;  
void thread_init(void) {  
    err = pthread_key_create(&key, destructor);  
}  
int threadfunc(void *arg) {  
    pthread_once(&init_done, thread_init);  
}
```

Ассоциация ключа с локальными данными

```
/* вернуть указатель на область памяти с локальными  
 * данными (или NULL) */
```

```
void *pthread_getspecific(pthread_key_t key);
```

```
/* связать ключ с локальными данными */
```

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

Локальные данные потоков - пример

```
pthread_key_t key;

void *foo(void *arg) {
    int *newdata = (int *) pthread_getspecific(key);

    if (newdata == NULL) {
        newdata = (int *) malloc(sizeof(int));
        int *argval = (int *) arg;
        *newdata = *argval;
        pthread_setspecific(key, newdata);
    }

    int *data = (int *) pthread_getspecific(key);
    printf("data = %d\n", *data);
    return NULL;
}
```

Локальные данные потоков - пример

```
int main(int argc, const char *argv[]) {  
    pthread_t tid1, tid2;  
    int arr[2] = {1234, 5678};  
  
    pthread_key_create(&key, NULL);  
  
    pthread_create(&tid1, NULL, foo, &arr[0]);  
    pthread_create(&tid2, NULL, foo, &arr[1]);  
  
    pthread_join(tid1, NULL);  
    pthread_join(tid2, NULL);  
  
    return 0;  
}
```


Локальные данные потоков - пример 2

```
static pthread_key_t key;
static pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_mutex_t env_mutex =
    PTHREADMUTEX_INITIALIZER;

extern char **environ;

static void thread_init(void)
{
    pthread_key_create(&key, free);
}

char *getenv(const char *name) {
    int i, len;
    char *envbuf;
```

Локальные данные потоков - пример 2

```
pthread_once(&init_done, thread_init);
pthread_mutex_lock(&env_mutex);
envbuf = (char *) pthread_getspecific(key);
if (envbuf == NULL) {
    envbuf = malloc(ARGV_MAX);
    if (envbuf == NULL) {
        pthread_mutex_unlock(&env_mutex);
        return NULL;
    }
    pthread_setspecific(key, envbuf);
}
```

Локальные данные потоков - пример 2

```
len = strlen(name);  
for (i = 0; environ[i] != NULL; i++) {  
    if ((strncmp(name, environ[i], len) == 0) &&  
        (environ[i][len] == '=')) {  
        strcpy(envbuf, &environ[i][len+1]);  
        pthread_mutex_unlock(&env_mutex);  
        return envbuf;  
    }  
}  
  
pthread_mutex_unlock(&env_mutex);  
return NULL;  
}
```

Потоки и сигналы

Потоки и сигналы

- Каждый поток имеет собственную маску потоков.
- Отдельно взятый поток может заблокировать доставку сигнала.
- Когда поток назначает обработчик сигнала, он становится общим для всех потоков. То есть сигналы могут “противоречить” друг другу.
- Сигнал доставляется только одному потоку. Если сигнал связан с аппаратной ошибкой или таймером, то доставляется потоку, который стал причиной. Остальные сигналы могут доставляться произвольному потоку.

Задать маску сигналов для потока

```
/* задать маску обработчиков сигналов */  
int pthread_sigmask(int how, const sigset_t *set,  
                    sigset_t *oldset);
```

- если `oldset != NULL`, в него возвращается текущая маска сигналов
- если в `set` передается непустой указатель, то значение `how` определяет, как должна измениться маска сигналов
- `how == SIG_BLOCK` - новая маска - объединение текущей маски и с набором, указанным в `set`
- `how == SIG_UNBLOCK` - новая маска - пересечение текущей маски с набором, на который указывает `set`
- `how == SIG_SETMASK` - новая маска `set` замещает текущую маску

Ожидания поступления сигнала потоком

```
/* приостановиться в ожидании хотя бы одного  
 * сигнала из set, сохранить поступивший сигнал  
 * в sig */  
int sigwait(const sigset_t *set, int *sig);
```

- Сигнал удаляется из набора сигналов, ожидающих обработки.
- Во избежании ошибочной обработки, поток должен заблокировать (pthread_sigmask) сигналы перед вызовом sigwait. Иначе сигнал может быть доставлен
- sigwait позволяет синхронно обрабатывать асинхронные сигналы
- Можно выделить **отдельные потоки, которые будут заниматься только обработкой сигналов**

Передача сигнала потоку

```
/* приостановиться в ожидании хотя бы одного  
 * сигнала из set, сохранить поступивший сигнал  
 * в sig */  
int sigwait(const sigset_t *set, int *sig);
```

```
/* передать сигнал потоку */  
int pthread_kill(pthread_t thread, int sig);
```

- Можно проверить существование потока, передав `sig = 0`
- Если действие по умолчанию для сигнала - завершение процесса, то передача сигнала потоку приведёт к завершению всего процесса
- n.b. таймеры являются ресурсами процесса

Потоки и сигналы - пример

```
int quitflag;  /* поток записывает ненулевое значение */
sigsetjt mask;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t wait = PTHREAD_COND_INITIALIZER;

void *thr_fn(void *arg) {
    int err, signo;
    for (;;) {
        err = sigwait(&mask, &signo);
        if (err != 0) {
            fprintf(stderr, "sigwait failed"); exit(1); }
    }
```

Потоки и сигналы - пример

```
switch (signo) {
    case SIGINT:
        printf("interruption");
        break;
    case SIGQUIT:
        pthread_mutex_lock(&lock);
        quitflag = 1;
        pthread_mutex_unlock(&lock);
        pthread_cond_signal(&wait);
        return 0;
    default:
        fprintf(stderr, "undef. signal %d\n", signo);
        exit(1)
}
}}
```

Потоки и сигналы - пример

```
int main(void) {
    int err; sigset_t oldmask; pthread_t tid;
    sigemptyset(&mask);
    sigaddset(&mask, SIGINT); sigaddset(&mask, SIGQUIT);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask,
                               &oldmask)) != 0) {
        fprintf(stderr, "SIG_BLOCK failed"); exit(1); }
    err = pthread_create(&tid, NULL, thr_fn, 0);
    if (err != 0) {
        fprintf(stderr, "can't create thread"); exit(1); }
    pthread_mutex_lock(&lock);
    while (quitflag == 0)
        pthread_cond_wait(&wait, &lock);
    pthread_mutex_unlock(&lock);
}
```

Потоки и сигналы - пример

```
/* Сигнал SIGQUIT был перехвачен и сейчас
 * опять заблокирован. */
quitflag = 0;

/* Восстановить маску сигналов, в которой
 * SIGQUIT разблокирован. */
if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
    fprintf(stderr, "SIG_SETMASK failed");
    exit(0);
}
}
```

Принудительное завершение потоков

Принудительное завершение потоков

```
/* установить атрибут принудительного завершения  
 * oldstate - старое значение атрибута */  
int pthread_setcancelstate(int state,  
                           int *oldstate);
```

PTHREAD_CANCEL_ENABLE - вызов `pthread_cancel` приводит к тому, что поток продолжает работу до **точки выхода**. Точки выхода определены стандартом.

PTHREAD_CANCEL_DISABLE - вызов `pthread_cancel` не приводит к завершению потока.

```
/* задать точку выхода из потока: если есть  
 * ожидающий запрос, то поток завершит работу */  
void pthread_testcancel(void);
```

Тип завершения потока

```
/* задать тип выхода */
```

```
int pthread_setcanceltype(int type, int *oldtype);
```

PTHREAD_CANCEL_DEFERRED - отложенный выход (до точки выхода)

PTHREAD_CANCEL_ASYNCHRONOUS - немедленный выход из потока при поступлении запроса

Тип завершения потока

```
/* задать тип выхода */
```

```
int pthread_setcanceltype(int type, int *oldtype);
```

PTHREAD_CANCEL_DEFERRED - отложенный выход (до точки выхода)

PTHREAD_CANCEL_ASYNCHRONOUS - немедленный выход из потока при поступлении запроса



Действия потоков при принудительном завершении

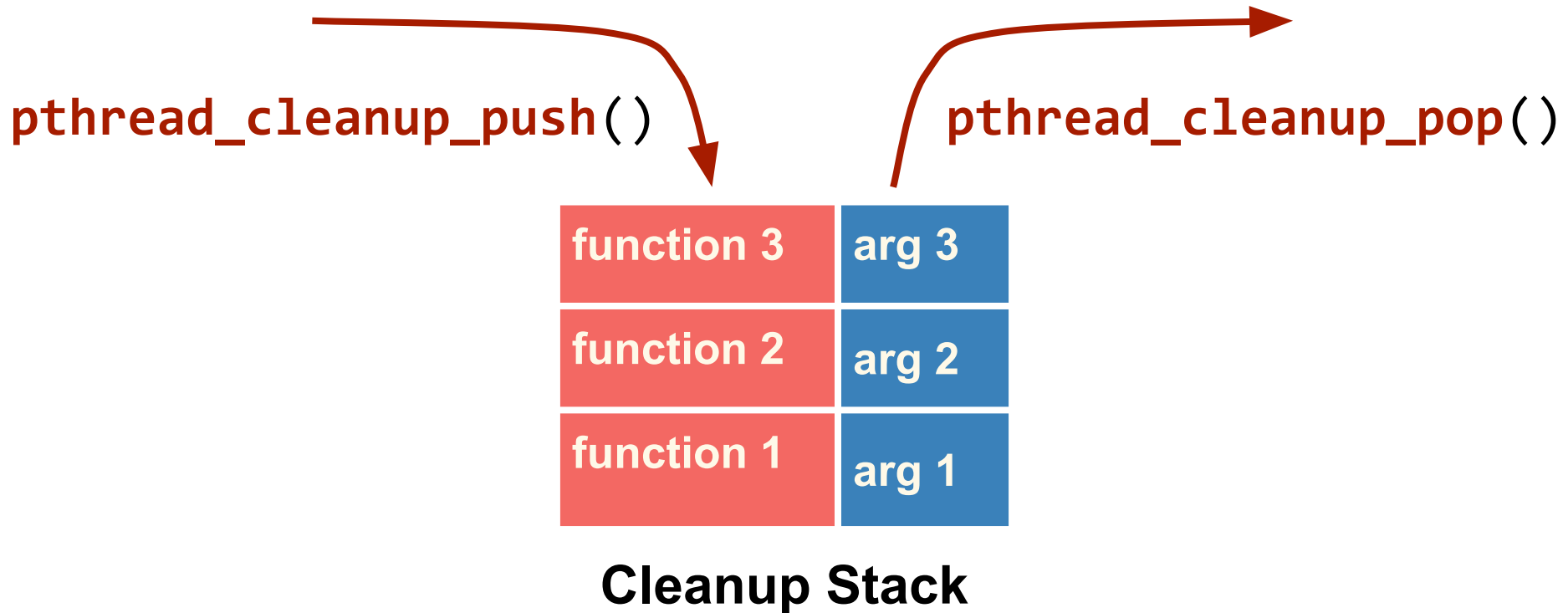
При завершении потока необходимо убедиться:

- Поток должен **освободить все блокировки**, которые он удерживает.
- Поток должен **освободить выделенную память**.
- Поток должен завершиться **в корректном состоянии**.



Асинхронная отмена может привести к нежелательным последствиям!

Обработчики отмены потока (cancellation cleanup handlers)



- Обработчики позволяют очистить состояние потока: выполнить какие-то действия при отмене.
- Функции 3-1 запускаются при отмене потока или срабатывания `pthread_exit`.

Обработчики отмены потока (cancellation cleanup handlers)

- `pthread_cleanup_push` и `pthread_cleanup_pop` ДОЛЖНЫ ВЫЗЫВАТЬСЯ **на одном уровне**.

```
pthread_cleanup_push(free, pointer);  
...  
pthread_cleanup_pop(1);
```

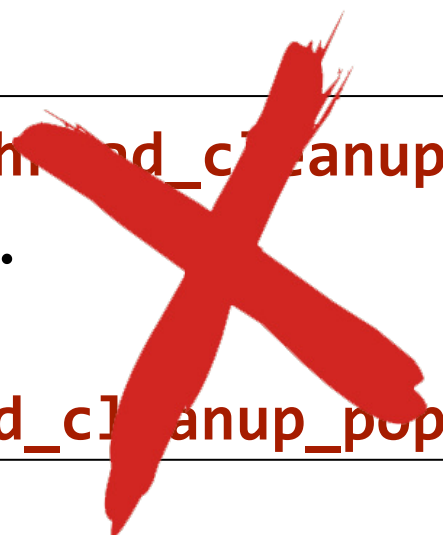
```
    pthread_cleanup_push(free, pointer);  
    ...  
}  
pthread_cleanup_pop(1);
```

Обработчики отмены потока (cancellation cleanup handlers)

- `pthread_cleanup_push` и `pthread_cleanup_pop` ДОЛЖНЫ ВЫЗЫВАТЬСЯ **на одном уровне**.

```
pthread_cleanup_push(free, pointer);  
...  
pthread_cleanup_pop(1);
```

```
pthread_cleanup_push(free, pointer);  
...  
}  
pthread_cleanup_pop(1);
```



Обработчики отмены потока - пример 2

```
pthread_cleanup_push(pthread_mutex_unlock,  
                    (void *) &mut);  
pthread_mutex_lock(&mut);  
/* do some work */  
pthread_mutex_unlock(&mut);  
pthread_cleanup_pop(0);
```

```
void unlock(void *arg) {  
    pthread_mutex_unlock( &lock );  
}  
void *func(void *arg) {  
    pthread_mutex_lock(&lock);  
    pthread_cleanup_push( &unlock, NULL );  
    /* ... */  
    pthread_cleanup_pop(1);
```

Пример (сигналы, отмена потоков)

```
enum { NTHREADS = 8 };

pthread_mutex_t file_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t nalive_lock =

PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t alldead_cond =
PTHREAD_COND_INITIALIZER;
pthread_once_t init_done = PTHREAD_ONCE_INIT;
pthread_key_t key_name;
int nalive = 0;
FILE *file;

sigset_t mask;

pthread_t tid[NTHREADS];

#define FILENAME "play"
```

Пример (сигналы, отмена потоков)

```
int main(int argc, const char *argv[]) {
    sigemptyset(&mask);
    sigaddset(&mask, SIGQUIT);
    sigset_t oldmask;
    int rc = pthread_sigmask(SIG_BLOCK, &mask, &oldmask);
    file = fopen(FILENAME, "w");
    fclose(file);
    pthread_t thr_reaper;
    pthread_create(&thr_reaper, NULL, grim_reaper, NULL);
    pthread_create(&tid[0], NULL, character, "Hamlet");
    pthread_create(&tid[1], NULL, character, "Ophelia");
    pthread_create(&tid[2], NULL, character, "Gertrude");
    pthread_create(&tid[3], NULL, character, "Claudius");
    pthread_create(&tid[4], NULL, character, "Polonius");
    pthread_create(&tid[5], NULL, character, "Laertes");
    pthread_create(&tid[6], NULL, character, "Rosencrantz");
    pthread_create(&tid[7], NULL, character, "Guildenstern");
}
```

Пример (сигналы, отмена потоков)

```
pthread_mutex_lock(&nalive_lock);
while (nalive > 0) {
    pthread_cond_wait(&alldead_cond, &nalive_lock);
}
pthread_mutex_unlock(&nalive_lock);
for (int tid_i = 0; tid_i < NTHREADS; tid_i++) {
    pthread_join(tid[tid_i], NULL);
}

file = fopen(FILENAME, "a");
fprintf(file, "The rest is silence.\n");
fclose(file);

pthread_cancel(thr_reaper);
pthread_join(thr_reaper, NULL);

return 0;
}
```


Пример (сигналы, отмена потоков)

```
void _sem_wait(sem_t *sem) {  
    while (sem_wait(sem) != 0) { }  
}  
  
void key_destructor(void *arg) {  
    printf("key destructor\n");  
    char *name = (char *) arg;  
    free(name);  
}  
  
void thread_init(void) {  
    pthread_key_create(&key_name, key_destructor);  
}
```

Пример (сигналы, отмена потоков)

```
void cleanup_lock(void *arg) {  
    printf("cleanup_lock\n");  
    pthread_mutex_t *lock = (pthread_mutex_t *) arg;  
    pthread_mutex_unlock(lock);  
}  
  
void cleanup_file(void *arg) {  
    printf("cleanup_file\n");  
    fclose(file);  
}
```

Пример (сигналы, отмена потоков)

```
void cleanup_death(void *arg) {  
    printf("cleanup_death\n");  
    pthread_mutex_lock(&file_lock);  
    file = fopen(FILENAME, "a");  
    fprintf(file, "%s is dead\n",  
            (char *) pthread_getspecific(key_name));  
    pthread_mutex_unlock(&file_lock);  
  
    pthread_mutex_lock(&nalive_lock);  
    if (--nalive == 0) {  
        pthread_mutex_unlock(&nalive_lock);  
        printf("The last character is dead.\n");  
        pthread_cond_signal(&alldead_cond);  
    }  
    pthread_mutex_unlock(&nalive_lock);  
}
```

Пример (сигналы, отмена потоков)

```
void *character(void *arg) {  
    pthread_once(&init_done, thread_init);  
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,  
NULL);  
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);  
    char *name = (char *) malloc(strlen(arg));  
    strcpy(name, arg);  
  
    pthread_setspecific(key_name, name);  
    pthread_mutex_lock(&nalive_lock);  
    nalive++;  
    pthread_mutex_unlock(&nalive_lock);  
}
```

Пример (сигналы, отмена потоков)

```
pthread_cleanup_push(cleanup_death, NULL);
for (;;) {
    fprintf(stderr, "I am %s\n", name);
    pthread_mutex_lock(&file_lock);
    pthread_cleanup_push(cleanup_lock, NULL);
    file = fopen(FILENAME, "a");
    pthread_cleanup_push(cleanup_file, NULL);
    fprintf(file, "%s was here\n",
            (char *) pthread_getspecific(key_name));
    pthread_cleanup_pop(0);
    fclose(file);
    pthread_cleanup_pop(0);
    pthread_mutex_unlock(&file_lock);
    pthread_testcancel();
    sleep(1);
}
pthread_cleanup_pop(0); return NULL; }
```

Пример (сигналы, отмена потоков)

```
void *grim_reaper(void *arg)
{
    for (;;) {
        int signo;
        int rc = sigwait(&mask, &signo);
        if (rc != 0) {
            fprintf(stderr, "sigwait() failed");
            exit(1);
        }
    }
}
```

Пример (сигналы, отмена потоков)

```
switch (signo) {
    case SIGQUIT:
        printf("SIGQUIT received\n");
        if (nalive > 0) {
            struct drand48_data rand48_buffer;
            srand48_r(time(NULL), &rand48_buffer);
            int rc;
            do {
                long int char_id;
                lrand48_r(&rand48_buffer, &char_id);
                char_id %= NTHREADS;
                rc = pthread_cancel(tid[char_id]);
            } while (rc == ESRCH);
        }
        break;
    default: printf("Unspec. signal\n"); exit(1);
} } } /* end of the function */
```

Пример (сигналы, отмена потоков)

```
$ ./signal_cleanup  
I am Rosencrantz and I am writing to the file  
I am Guildenstern and I am writing to the file  
I am Polonius and I am writing to the file  
I am Claudius and I am writing to the file  
I am Gertrude and I am writing to the file  
I am Ophelia and I am writing to the file  
I am Hamlet and I am writing to the file  
I am Laertes and I am writing to the file  
I am Rosencrantz and I am writing to the file  
I am Guildenstern and I am writing to the file  
I am Polonius and I am writing to the file  
I am Claudius and I am writing to the file  
I am Gertrude and I am writing to the file  
I am Ophelia and I am writing to the file  
I am Hamlet and I am writing to the file
```


Пример (сигналы, отмена потоков)

SIGQUIT received
cleanup_death
key destructor

```
kill -SIGQUIT signal_cleanup
```

```
I am Gertrude and I am writing to the file
I am Guildenstern and I am writing to the file
I am Ophelia and I am writing to the file
I am Laertes and I am writing to the file
I am Polonius and I am writing to the file
I am Hamlet and I am writing to the file
I am Rosencrantz and I am writing to the file
I am Gertrude and I am writing to the file
I am Guildenstern and I am writing to the file
I am Ophelia and I am writing to the file
I am Laertes and I am writing to the file
I am Hamlet and I am writing to the file
I am Polonius and I am writing to the file
```

Пример (сигналы, отмена потоков)

```
SIGQUIT received  
cleanup_death  
key destructor
```

```
pkill -SIGQUIT signal_cleanup
```

```
I am Gertrude and I am writing to the file
```

```
I am Laertes and I am writing to the file
```

```
I am Ophelia and I am writing to the file
```

```
I am Rosencrantz and I am writing to the file
```

```
I am Hamlet and I am writing to the file
```

```
I am Polonius and I am writing to the file
```

```
I am Gertrude and I am writing to the file
```

```
I am Laertes and I am writing to the file
```

```
I am Ophelia and I am writing to the file
```

```
I am Rosencrantz and I am writing to the file
```

```
I am Hamlet and I am writing to the file
```

```
I am Polonius and I am writing to the file
```

```
I am Rosencrantz and I am writing to the file
```

Пример (сигналы, отмена потоков)

SIGQUIT received
cleanup_death
key destructor

```
kill -SIGQUIT signal_cleanup
```

[illegible]

Пример (сигналы, отмена потоков)

SIGQUIT received
cleanup_death
key destructor

`pkill -SIGQUIT signal_cleanup`

I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file
I am Gertrude and I am writing to the file

SIGQUIT received
cleanup_death

`pkill -SIGQUIT signal_cleanup`

The last character is dead.
key destructor

Пример (сигналы, отмена потоков)

```
$ cat play
Laertes was here
Rosencrantz was here
Guildenstern was here
Polonius was here
Claudius was here
Gertrude was here
Polonius is dead
Hamlet was here
Laertes was here
...
Laertes was here
Gertrude was here
Hamlet was here
Ophelia is dead
Rosencrantz was here
```

Пример (сигналы, отмена потоков)

[illegible]

Шаблоны

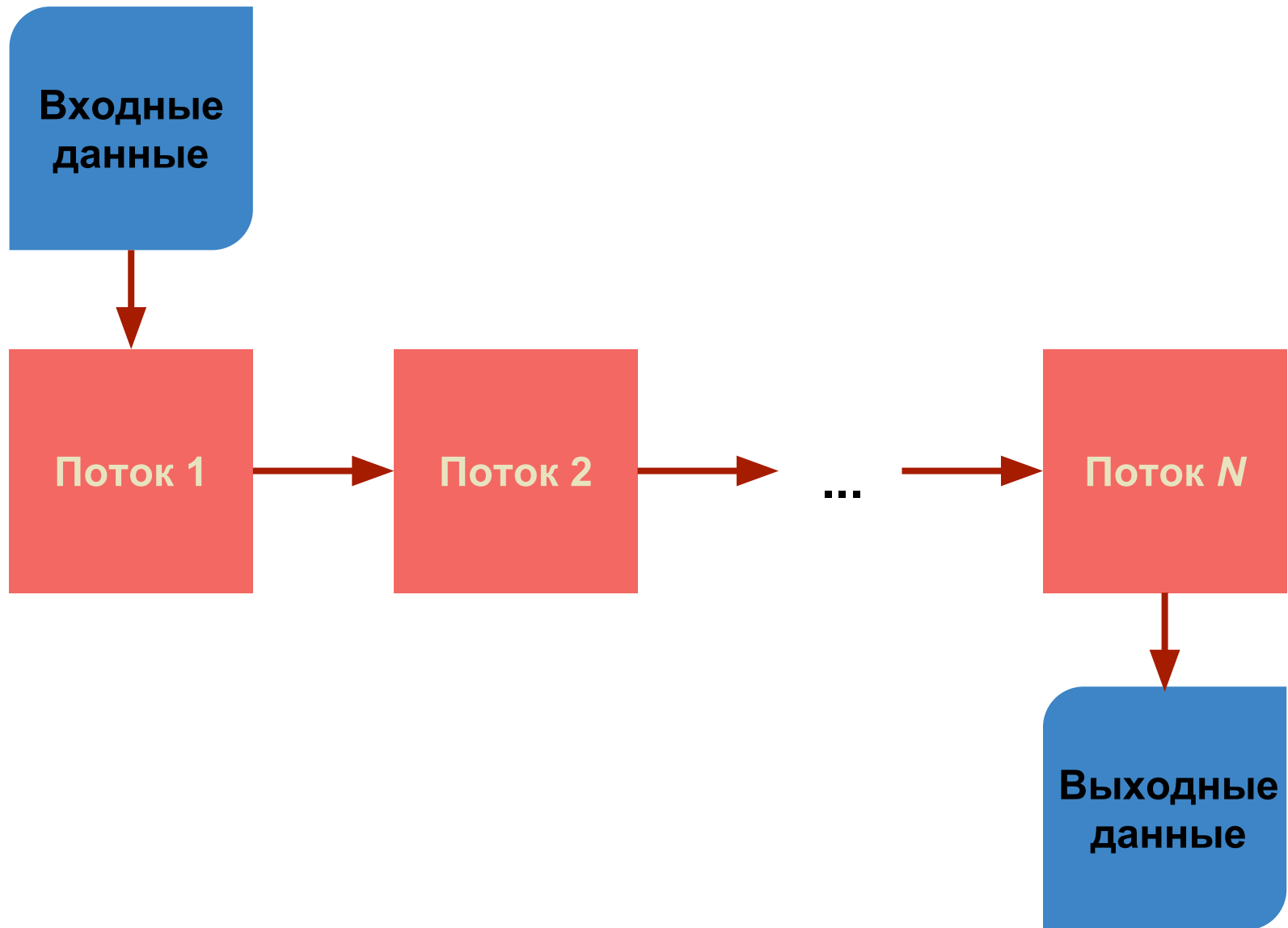
использования потоков

Паттерны использования потоков

- **Конвейер (pipeline)**. Каждый поток повторяет заданную операцию над очередной порцией данных (в рамках последовательности данных), затем отправляет её следующему потоку для выполнения следующего шага.
- **Группа (work crew)**. Каждый поток выполняет операцию над своими данными. Все потоки работают независимо.
- **Клиент/сервер (client/server)**. Клиенты взаимодействуют с сервером для решения каждой задачи. Задачи становятся в очередь, затем выполняются.
- **Очередь (work queue)**. Есть несколько потоков, которые перераспределяют задачи между собой с целью наибольшей загрузки процессоров.

В одной программе может использоваться несколько моделей.

Конвейер (pipeline)



Конвейер (pipeline)

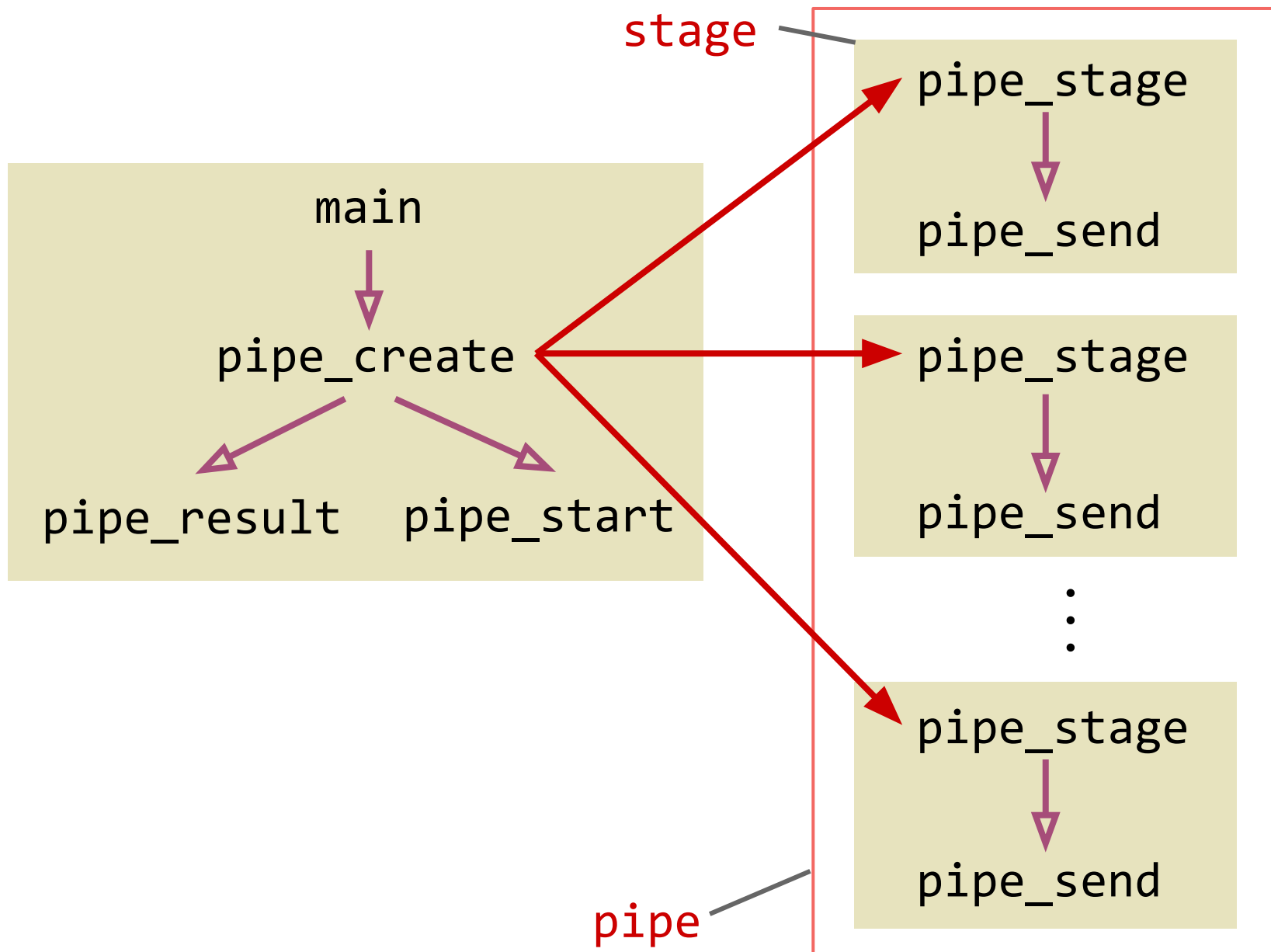
- Последовательность элементов данных последовательно обрабатывается потоками.
- Каждый поток выполняет определённую операцию.
- Ускорение достигается тем, что при достаточно длинной последовательности потоки обрабатывают данные одновременно.
- Модель MISD.

Конвейер (pipeline)

Пример

Каждый поток в конвейере увеличивает входное значение на 1 и передаёт его следующему потоку. Основная программа читает последовательность строк из потока ввода. Команда представляет собой последовательность чисел и знаков "=", которые заставляют конвейер читать с конца текущий результат и выводить в поток вывода.

Конвейер (pipeline) - пример



Конвейер: структуры данных

1. Стадия

- защита стадии
- ГОТОВНОСТЬ ДАННЫХ
- ПОТОК
- поддержка списка

2. Конвейер

- защита конвейера
- список стадий
- счётчики текущего состояния

Конвейер (pipeline) - пример

```
/*
 * Структура, описывающая стадию конвейера.
 * Для каждой такой стадии создаётся поток,
 * кроме последней стадии, которая хранит результат.
 */

typedef struct stage_tag {
    pthread_mutex_t mutex;        /* Защита данных */
    pthread_cond_t avail;        /* Данные доступны */
    pthread_cond_t ready;        /* Поток готов для
                                   * получения данных */
    int data_ready;              /* Флаг готовности данных */
    long data;                   /* Данные для обработки */
    pthread_t thread;            /* Поток для выполнения */
    struct stage_tag *next;      /* Следующая стадия */
} stage_t;
```

Конвейер (pipeline) - пример

```
/*  
 * Структура, описывающая весь конвейер.  
 */  
  
typedef struct pipe_tag {  
    pthread_mutex_t mutex;    /* Защита конвейера */  
    stage_t *head;           /* Первая стадия */  
    stage_t *tail;           /* Последняя стадия */  
    int stages;              /* Число стадий */  
    int active;              /* Активность конвейера */  
} pipe_t;
```

Конвейер: отправка данных в стадию

1. Если стадия ещё занята своими данными, подождать, пока не освободится.
2. Как освободилась, записать в её структуру данные.
3. Сообщить стадии, что данные ГОТОВЫ.

Конвейер: pipe_send(stage_t *stage, long data)

```
/* Отправить сообщение в указанный блок конвейера. */
int pipe_send (stage_t *stage, long data) {
    pthread_mutex_lock (&stage->mutex);

    /* Дождаться новой порции данных */
    while (stage->data_ready)
        pthread_cond_wait(&stage->ready, &stage->mutex);

    /* Отправить новые данные */
    stage->data = data;
    stage->data_ready = 1;
    pthread_cond_signal(&stage->avail);
    pthread_mutex_unlock (&stage->mutex);

    /* Вернуть код ошибки запуска функций pthread_... */
    return rc;
}
```

Конвейер: работа стадии

1. В бесконечном цикле ожидаем сообщения о готовности данных для него.
2. Пришёл запрос - обработать данные...
3. и отправить запрос следующей стадии.
4. Сообщить предыдущей стадии о том, что текущая готова для новой порции.

Конвейер: pipe_stage (void *arg)

```
/* Дождаться данных, обработать их
 * и отправить на следующую стадию */

void *pipe_stage (void *arg) {
    stage_t *stage = (stage_t*)arg;
    stage_t *next_stage = stage->next;
    pthread_mutex_lock (&stage->mutex);

    for (;;) {
        while (stage->data_ready != 1)
            pthread_cond_wait(&stage->avail, &stage->mutex);
        pipe_send (next_stage, stage->data + 1);
        stage->data_ready = 0;
        pthread_cond_signal(&stage->ready);
    }

    /* Почему функция не выполняет разблокировку
     * stage->mutex? */
}
```

Конвейер: создание конвейера

1. Проинициализировать список
 - последняя стадия - завершающая, без потока
2. Для каждой стадии
 - выделить структуру данных и проинициализировать
 - отдельно обработать последнюю стадию
3. Порождать поток для каждой стадии (кроме последней)

Конвейер: pipe_create (pipe_t *pipe, int stages)

```
/* Инициализация данных, создание потоков. */
int pipe_create (pipe_t *pipe, int stages) {
    int pipe_index;
    stage_t **link = &pipe->head, *new_stage, *stage;
    pthread_mutex_init (&pipe->mutex, NULL);
    pipe->stages = stages;
    pipe->active = 0;
    for (pipe_index = 0; pipe_index <= stages; pipe_index++) {
        new_stage = (stage_t*)malloc (sizeof(stage_t));
        pthread_mutex_init (&new_stage->mutex, NULL);
        rc = pthread_cond_init (&new_stage->avail, NULL);
        rc = pthread_cond_init (&new_stage->ready, NULL);
        new_stage->data_ready = 0;
        *link = new_stage;
        link = &new_stage->next;
    }
    *link = (stage_t *) NULL;
    pipe->tail = new_stage;
    /* Завершить список */
    /* Записать хвост списка */
}
```

Конвейер (pipeline)

```
/*
 * Потоки для стадий конвейера создаются только после того,
 * как все данные проинициализированы. Для последней
 * стадии не создаётся потока, т.к. она только для
 * хранения результата.
 *
 * В случае возникновения ошибок необходимо отменить
 * и отсоединить уже созданные потоки и удалить объекты
 * синхронизации
 */
for (stage = pipe->head; stage->next != NULL
     stage = stage->next) {
    rc = pthread_create(&stage->thread, NULL,
                       pipe_stage, (void *) stage);

    if (rc != 0)
        /* Обработка ошибок */
}
return 0;
}
```

Конвейер (pipeline)

```
/*  
 * Запустить конвейер, отправив данные на первую стадию.  
 */  
  
int pipe_start(pipe_t *pipe, long value) {  
    pthread_mutex_lock (&pipe->mutex);  
    pipe->active++;  
    pthread_mutex_unlock (&pipe->mutex);  
    pipe_send (pipe->head, value);  
    return 0;  
}
```

Конвейер: получить результат

1. Есть ли не полученные результаты в конвейере.
2. Если нет, вернуть 0.
3. Если есть, ожидать, пока не появится результат в последней стадии.
4. Как появился, взять его и сообщить, что последняя стадия готова для новой порции данных.

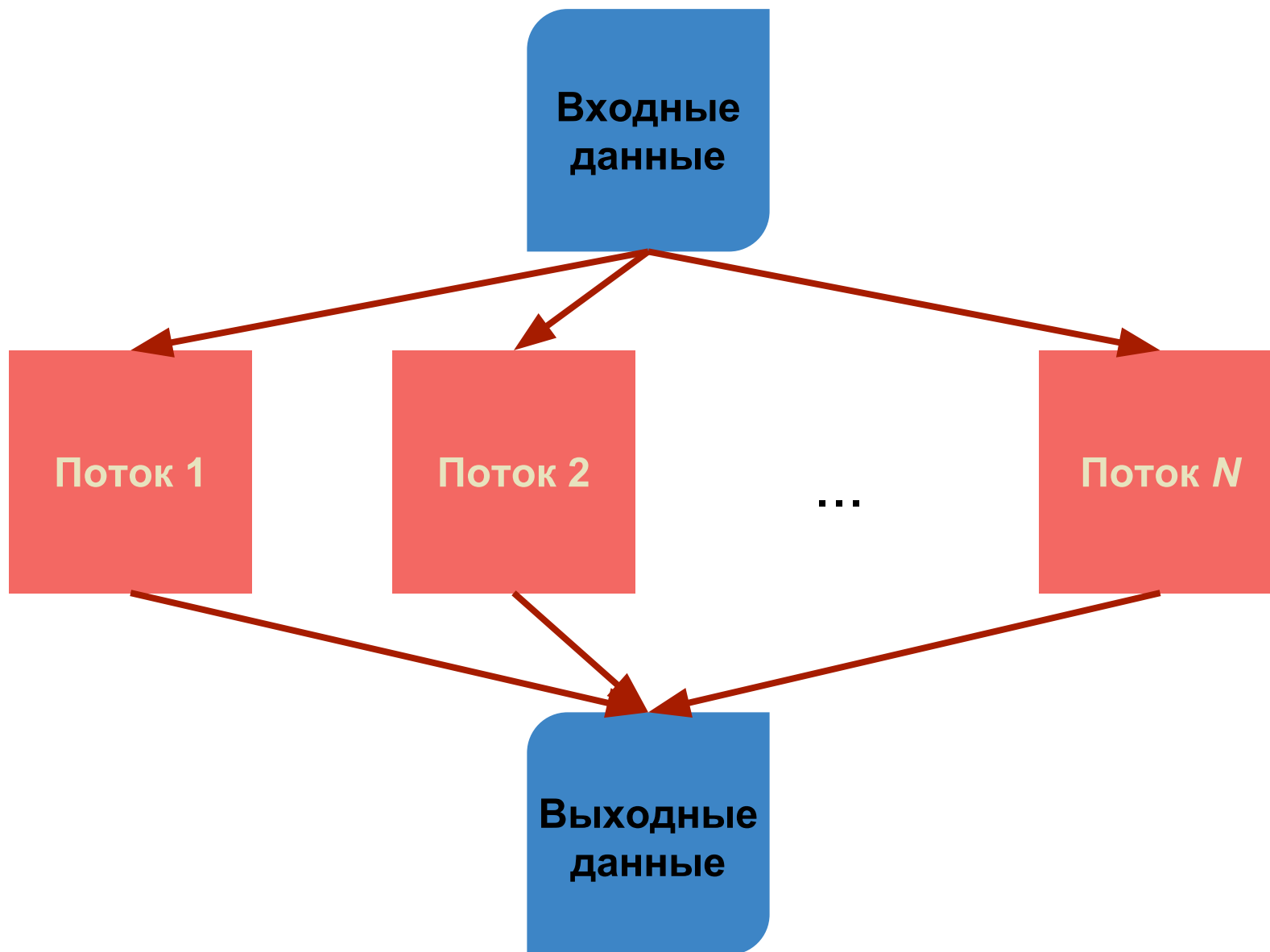
Конвейер (pipeline)

```
/* Забрать результат из потока.  
 * Дождаться, пока результат не будет получен. */  
int pipe_result (pipe_t *pipe, long *result) {  
    stage_t *tail = pipe->tail;  
    long value; int empty = 0, rc;  
  
    pthread_mutex_lock (&pipe->mutex);  
    if (pipe->active <= 0) { empty = 1; }  
    else pipe->active--;  
    pthread_mutex_unlock (&pipe->mutex);  
  
    if (empty) { return 0; }  
  
    pthread_mutex_lock (&tail->mutex);  
    while (!tail->data_ready)  
        pthread_cond_wait (&tail->avail, &tail->mutex);  
    *result = tail->data;  
    tail->data_ready = 0;  
    pthread_cond_signal (&tail->ready);  
    pthread_mutex_unlock (&tail->mutex);  
    return 1; }
```

Конвейер (pipeline)

```
int main (int argc, char *argv[]) {
    pipe_t my_pipe;
    long value, result; int status; char line[128];
    pipe_create (&my_pipe, 10);
    printf ("Enter int values, or \"=\" for next result\n");
    for (;;) {
        if (fgets (line, sizeof(line), stdin) == NULL) exit (0);
        if (strlen (line) <= 1) continue;
        if (strlen (line) <= 2 && line[0] == '=') {
            if (pipe_result(&my_pipe, &result))
                printf ("Result is %ld\n", result);
            else
                printf ("Pipe is empty\n");
        } else {
            if (sscanf (line, "%ld", &value) < 1)
                fprintf (stderr, "Enter an integer value\n");
            else
                pipe_start (&my_pipe, value);
        }
    }
}
```

Группа потоков (work crew)



Группа потоков (work crew)

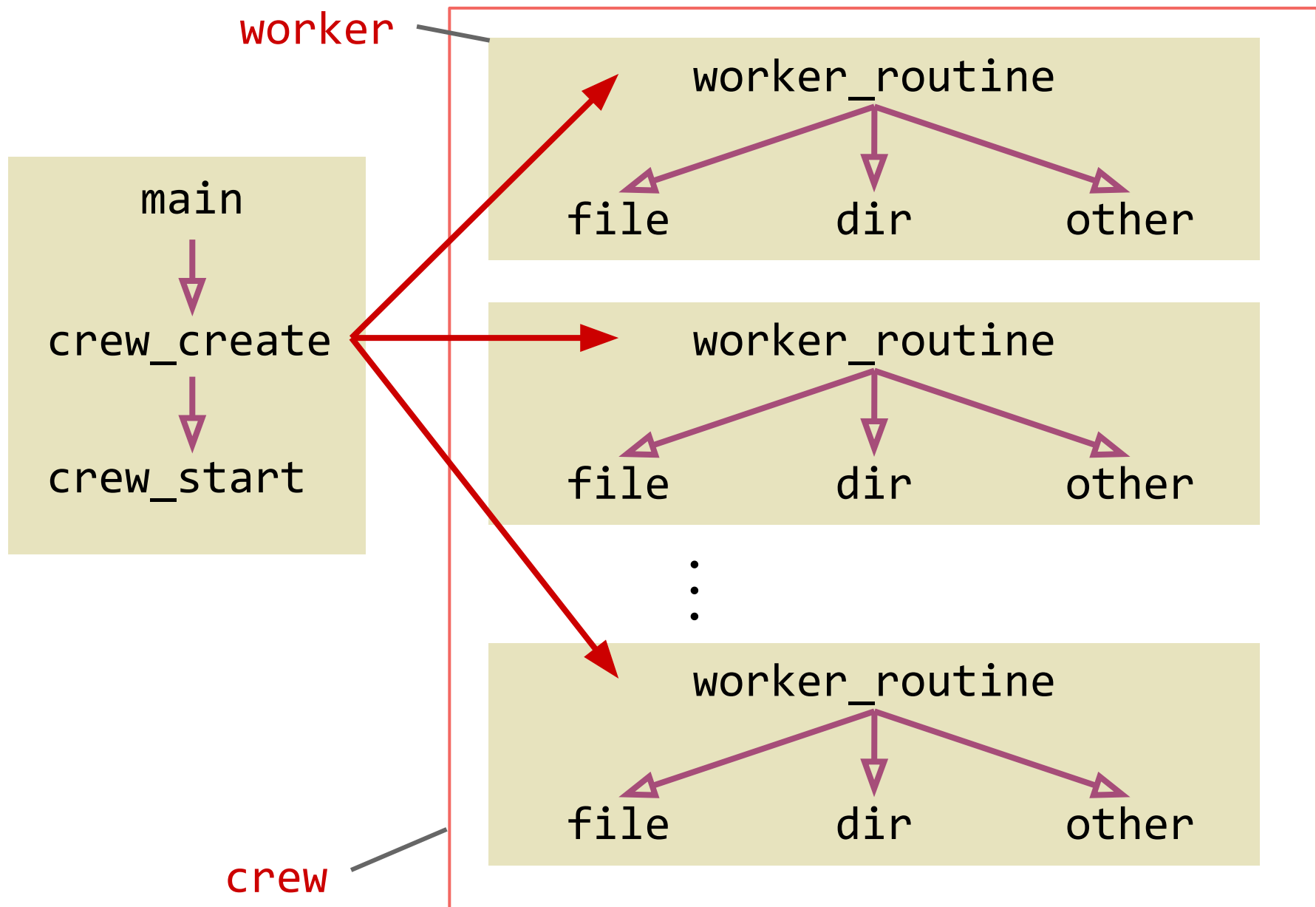
- Данные обрабатываются **независимо** набором потоков.
- Модель **SIMD**.
- В общем случае потоки могут выполнять **различные операции** над различными данными.
- Например, потоки могут забирать порции данных из глобальной очереди заданий и выполнять с ними различными операциями. Наличие **глобальной очереди** - отличие work crew от независимых потоков.

Группа потоков (work crew) - пример

Пример

Программа запускается с двумя параметрами: путь к файлу и строка. Путь добавляет путь как новое задание в очередь для группы потоков. Каждый участник группы определяет, является ли путь файлом или директорией. Если файл, то он ищет в нём строку. Если директория, он находит все поддиректории и обычные файлы и затем помещает их в очередь группы потоков в качестве новых заданий. Все файлы, которые содержат заданную строку, выводятся.

Группа потоков (work crew) - пример



Группа потоков: структуры данных

1. Задание (единица работы)

- поддержка списка
- само задание (данные)

2. Рабочий поток

- указатель на группу
- поток
- номер

3. Группа потоков

- массив рабочих потоков
- список заданий
- синхронизация

Группа потоков (work crew) - пример

```
#define CREW_SIZE 4

/* Элементы, которые становятся в очередь на выполнение
 * группой. Каждый такой элемент помещается в очередь
 * в crew_start и каждый поток может добавлять элементы.*/
typedef struct work_tag {
    struct work_tag *next;      /* Следующее задание */
    char *path;                 /* Путь (директория или файл) */
    char *string;               /* Строка для поиска */
} work_t, *work_p;

/* Структура определяется для каждого рабочего потока
 * и идентифицирует его. */
typedef struct worker_tag {
    int index;                  /* Индекс потока */
    pthread_t thread;
    struct crew_tag *crew;     /* Указатель на группу */
} worker_t, *worker_p;
```


Группа потоков (work crew) - пример

```
/* Обработчик группы потоков.
 * Содержит средства синхронизации работы группы. */

typedef struct crew_tag {
    int            crew_size;           /* Размер массива */
    worker_t      crew[CREW_SIZE];     /* Члены группы */
    long          work_count;          /* Число рабочих потоков */
    work_t        *first, *last;       /* Первый и последний
                                     * потоки в группе */
    pthread_mutex_t mutex;             /* Мьютекс для данных */
    pthread_cond_t done;              /* Условие завершения
                                     * работы группы */
    pthread_cond_t go;                /* Условие начала
                                     * работы группы */
} crew_t, *crew_p;

size_t path_max;                    /* Длина пути к файлу */
size_t name_max;                   /* Длина имени файла */
```

Группа потоков: рабочий поток

1. Выделение памяти под буфер для путей.
2. Ждём, когда поступит задание (путь+строка).
3. Как появилось, начинается бесконечный цикл:
 - ждём, пока не появится задание
 - появилось - удаляем его из очереди
 - определяем, чем является путь в этом задании:
 - если файл, ищем строку и выводим найденное
 - если директория, ищем все пути в этой директории и добавляем их как новые задания в очередь заданий
4. Освобождаем выделенную память, уменьшаем число задач.
5. Если больше нет задач, сообщаем об этом всем потокам в группе.

Группа потоков: worker_routine (void *arg)

```
/* Поток выполняет обработку одного элемента в группе.
 * Ждать до поступления "go" и обрабатывать запросы
 * до команды shut down. */
void *worker_routine (void *arg) {
    worker_p mine = (worker_t *) arg;
    crew_p crew = mine->crew;
    work_p work, new_work;
    struct stat filestat;
    struct dirent *entry; int rc;

    /* Выделить большой блок данных
     * и использовать его как буфер.*/
    entry = malloc (sizeof(struct dirent) + name_max);

    pthread_mutex_lock (&crew->mutex);
    while (crew->work_count == 0) {
        pthread_cond_wait (&crew->go, &crew->mutex);
    }
    pthread_mutex_unlock (&crew->mutex);
```

Группа потоков: worker_routine (void *arg)

```
/* Пока есть задания, выполнять */
for (;;) {
    /* Ждать, пока не появятся задания */
    pthread_mutex_lock (&crew->mutex);
    while (crew->first == NULL) {
        pthread_cond_wait (&crew->go, &crew->mutex);
    }

    /* Удалить задачу из очереди и выполнить */
    work = crew->first;
    crew->first = work->next;
    if (crew->first == NULL)
        crew->last = NULL;
    pthread_mutex_unlock (&crew->mutex);
    lstat (work->path, &filestat);
}
```

Группа потоков: worker_routine (void *arg)

```
if (S_ISLNK (filestat.st_mode))
    printf("Thread %d: %s is a link, skipping.\n",
           mine->index, work->path);
else if (S_ISDIR (filestat.st_mode)) {
    DIR *directory;
    struct dirent *result;
    /* Если директория, найти все файлы
       * и поставить их в очередь как новые задания. */
    directory = opendir (work->path);
    if (directory == NULL) {
        fprintf(stderr, "...");
        continue;
    }
}
```

Группа потоков: worker_routine (void *arg)

```
for (;;) {
    rc = readdir_r (directory, entry, &result);
    if (rc != 0) { fprintf(stderr, "..."); break; }
    if (result == NULL)
        break; /* Конец директории */

    /* Игнорировать "." и ".." */
    if (strcmp (entry->d_name, ".") == 0) continue;
    if (strcmp (entry->d_name, "..") == 0) continue;

    new_work = (work_p) malloc (sizeof (work_t));
    new_work->path = (char*) malloc (path_max);
    strcpy (new_work->path, work->path);
    strcat (new_work->path, "/");
    strcat (new_work->path, entry->d_name);
    new_work->string = work->string;
    new_work->next = NULL;
```

Группа потоков: worker_routine (void *arg)

```
/* Добавляем задание (файл) в очередь */  
pthread_mutex_lock (&crew->mutex);  
if (crew->first == NULL) {  
    crew->first = new_work;  
    crew->last = new_work;  
} else {  
    crew->last->next = new_work;  
    crew->last = new work;  
}  
crew->work_count++;  
pthread_cond_signal (&crew->go);  
pthread_mutex_unlock (&crew->mutex);  
} /* Конец внутреннего цикла for (;;) */  
closedir(directory);
```

Группа потоков: worker_routine (void *arg)

```
} else if (S_ISREG (filestat.st_mode)) {  
    FILE *search;  
    char buffer[256], *bufptr, *search_ptr;  
    /* Если это файл, то открываем и ищем строку */  
    search = fopen (work->path, "r");  
    for (;;) {  
        bufptr = fgets(buffer, sizeof (buffer), search);  
        if (bufptr == NULL) {  
            if (feof (search)) break;  
            if (ferror(search)) { fprintf("..."); break; }  
        }  
        search_ptr = strstr(buffer, work->string);  
        if (search_ptr != NULL) {  
            printf("Thread %d found \"%s\" in %s\n",  
                mine->index, work->string, work->path);  
            break;  
        }  
    }  
    fclose(search);  
}
```


Группа потоков: worker_routine (void *arg)

```
} else
    /* Если не файл и не директория, пишем сообщение */
    fprintf(stderr, /* ... */);
free(work->path);      /* Освобождаем буфер для пути */
free(work);            /* и память для задачи */
/* Уменьшаем число заданий и будим ожидающих выходных
 * данных, если группа задач оказалась пуста. */
status = pthread_mutex_lock (&crew->mutex);
crew->work count--;
if (crew->work_count <= 0) {
    pthread_cond_broadcast (&crew->done);
    pthread_mutex_unlock (&crew->mutex);
    break;
}
pthread_mutex_unlock (&crew->mutex);
}
free (entry);
return NULL;
}
```

Группа потоков: worker_routine (void *arg)

```
/* Уменьшаем число заданий и будим ожидающих выходных
 * данных, если группа задач оказалась пуста. */
status = pthread_mutex_lock (&crew->mutex);
crew->work count--;
if (crew->work_count <= 0) {
    pthread_cond_broadcast (&crew->done);
    pthread_mutex_unlock (&crew->mutex);
    break;
}
pthread_mutex_unlock (&crew->mutex);
}

free (entry);
return NULL;
}
```

Группа потоков: crew_create (crew_t *crew, int crew_size)

```
int crew_create (crew_t *crew, int crew_size) {
    int crew_index;
    if (crew_size > CREW_SIZE) return EINVAL;
    crew->crew_size = crew_size;
    crew->work_count = 0;
    crew->first = crew->last = NULL;

    /* Инициализировать объекты синхронизации */
    pthread_mutex_init (&crew->mutex, NULL);
    pthread_cond_init (&crew->done, NULL);
    pthread_cond_init (&crew->go, NULL);

    /* Создать рабочие потоки */
    for (crew_index = 0; crew_index < CREW_SIZE; crew_index++){
        crew->crew[crew_index].index = crew_index;
        crew->crew[crew_index].crew = crew;
        pthread_create(&crew->crew[crew_index].thread,
            NULL, worker_routine, (void*)&crew->crew[crew_index]);
    }
    return 0; }
```

Группа потоков: запуск группы

1. Ждём, пока группа закончит выполнение предыдущего (глобального) задания.
2. Создаём и инициализируем структуру под запрос.
3. Инициализируем список запросов для группы, добавляем туда новый запрос.
4. Сообщаем группе о начале выполнения.
5. Ждём завершения работы группы.

Группа потоков: crew_start(crew_p crew, char *filepath, ...

```
/* Отправить путь в группу потоков, предварительно созданную
 * с использованием crew_create */
int crew_start(crew_p crew, char *filepath, char *search) {
    work_p request;
    pthread_mutex_lock (&crew->mutex);
    /* Если группа занята, дождаться её завершения. */
    while (crew->work_count > 0)
        pthread_cond_wait (&crew->done, &crew->mutex);

    path_max = pathconf (filepath, PC_PATH_MAX);
    if (path_max == -1) { /* ... */ }
    name_max = pathconf(filepath, _PC_NAME_MAX);
    if (name_max == -1) { /* ... */ }
    path_max++; name_max++;

    request = (work_p) malloc(sizeof(work_t));
    request->path = (char *) malloc(path_max);
    strcpy (request->path, filepath);
    request->string = search;
    request->next = NULL;
```

Группа потоков: crew_start(crew_p crew, char *filepath, ...

```
if (crew->first == NULL) {
    crew->first = request;
    crew->last = request;
} else {
    crew->last->next = request;
    crew->last = request;
}
crew->work_count++;
pthread_cond_signal (&crew->go);
while (crew->work_count > 0)
    pthread_cond_wait (&crew->done, &crew->mutex);
pthread_mutex_unlock (&crew->mutex);
return 0;
}
```

Группа потоков: `crew_start(crew_p crew, char *filepath, ...`

```
if (crew->first == NULL) {
    crew->first = request; crew->last = request;
} else {
    crew->last->next = request; crew->last = request;
}
crew->work_count++;
rc = pthread_cond_signal (&crew->go);
if (rc != 0) {
    free (crew->first);
    crew->first = NULL;
    crew->work count = 0;
    pthread_mutex_unlock (&crew->mutex);
    return rc;
}
while (crew->work_count > 0) {
    rc = pthread_cond_wait (&crew->done, &crew->mutex);
    if (rc != 0)
        error("Waiting for crew to finish");
}
```

**Обработка
ошибок!**

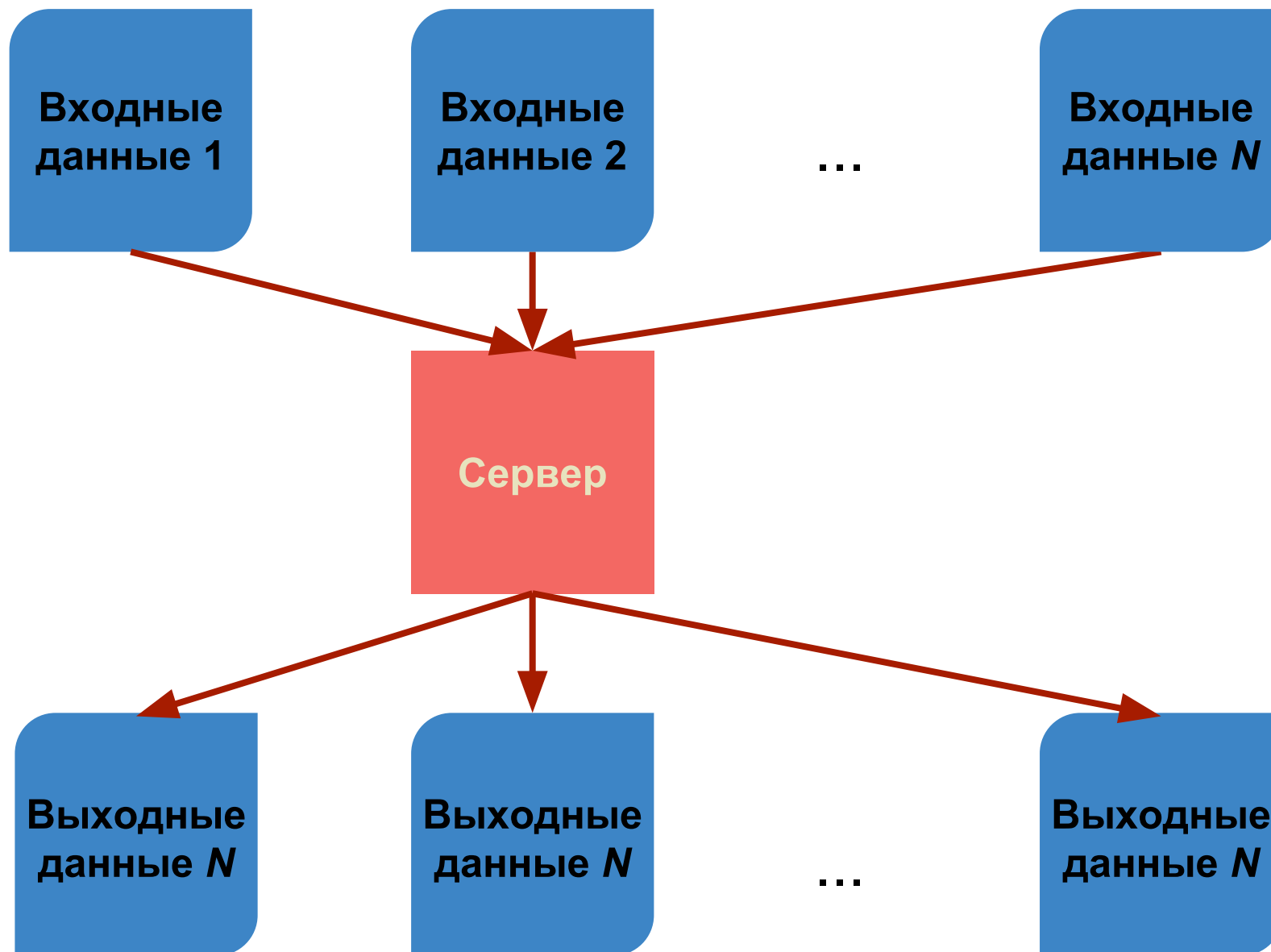
Группа потоков: main

```
int main (int argc, char *argv[]) {
    crew_t my_crew;
    char line[128], *next;
    int rc;
    if (argc < 3) {
        fprintf (stderr, "Usage: %s string path\n", argv[0]);
        return -1;
    }

    rc = crew_create (&my_crew, CREW_SIZE);
    if (status != 0)
        error("Create crew");

    rc = crew_start (&my_crew, argv[2], argv[1]);
    if (rc != 0)
        error("Start crew");
    return 0;
}
```


Клиент/сервер (client/server)



Клиент/сервер (client/server)

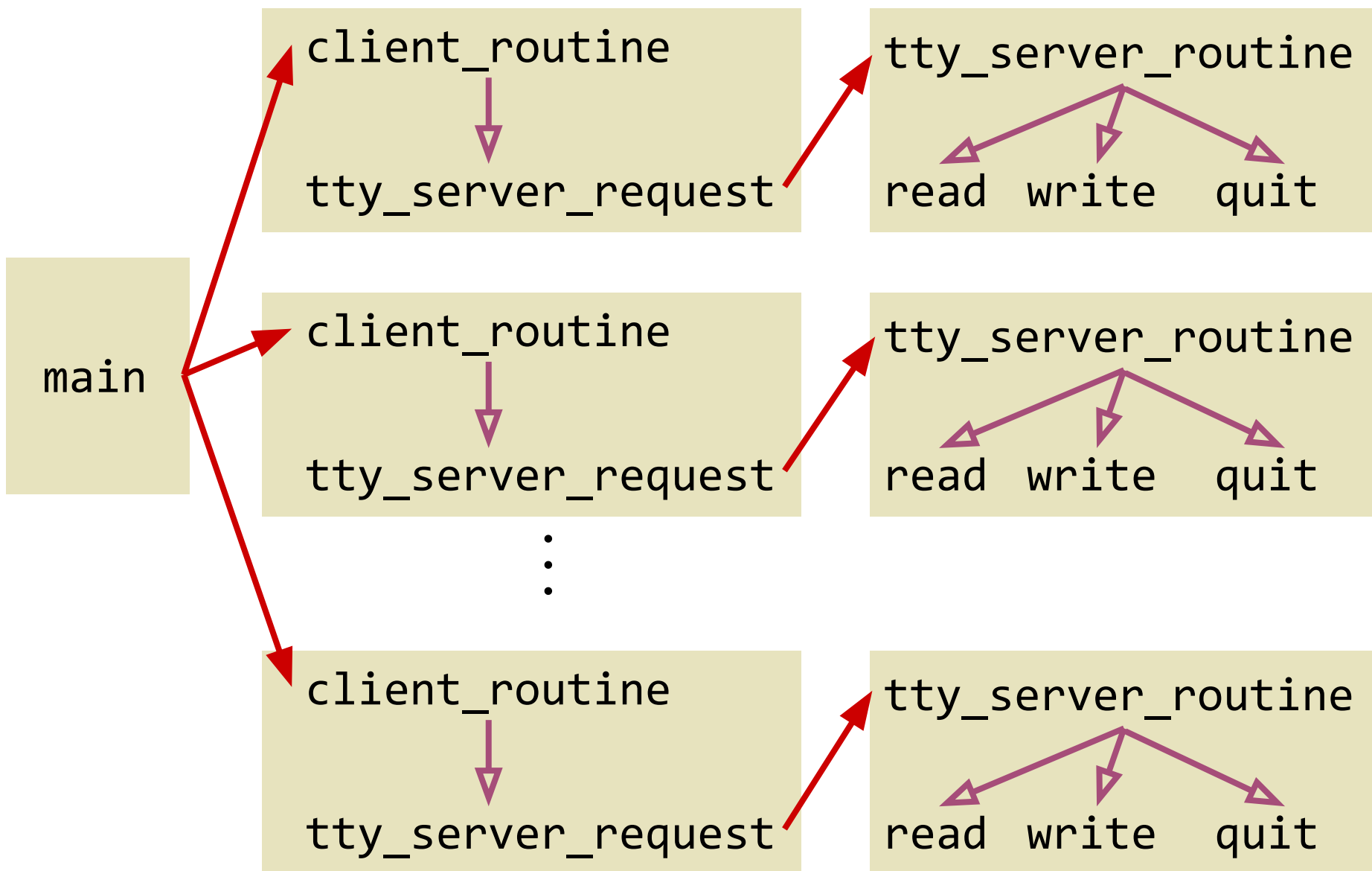
- Клиент запрашивает у сервера выполнение некоторой операции над данными.
- Сервер выполняет операцию независимо - клиент может как ждать сервера (синхронный вариант) или уснуть и запросить результат позже (асинхронный вариант).
- **Асинхронная модель** используется значительно чаще.

Клиент/сервер (client/server) - пример

Пример

Четыре потока независимо друг от друга читают и выводят строки. Для этого они отправляют запросы серверу, который выполняет чтение или вывод информации.

Конвейер (pipeline) - пример



Клиент/сервер (client/server) - структуры данных

1. Пакет для передачи запроса серверу:

- поддержка списка
- параметры
- данные
- синхронизация

2. Состояние сервера

- список пакетов
- синхронизация

3. Синхронизация клиентов

Клиент/сервер (client/server) - пример

```
#define CLIENT_THREADS 4    /* Число клиентов */

#define REQ_READ 1         /* Читать с подтверждением */
#define REQ_WRITE 2        /* Писать */
#define REQ_QUIT 3         /* Завершение работы сервера */

/* Пакет, создаваемый для каждого запроса серверу */
typedef struct request_tag {
    struct request_tag *next; /* Указатель на след. запрос */
    int operation;            /* Код операции*/
    int synchronous;          /* Синхронный режим */
    int done_flag;            /* Флаг ожидания завершения */
    pthread_cond_t done;      /* Условие завершения */
    char prompt[32];          /* Строка подтверждения */
    char text[128];           /* Текст для чтения/записи */
} request_t;
```

Клиент/сервер (client/server) - пример

```
/* Состояние сервера */
typedef struct tty_server_tag {
    request_t *first;
    request_t *last;
    int running;
    pthread_mutex_t mutex;
    pthread_cond_t request;
} tty_server_t;

/* Инициализация состояния сервера */
tty_server_t tty_server = {
    NULL, NULL, 0,
    PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER
};

/* Данные для основной программы */
int client_threads;
pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t clients_done = PTHREAD_COND_INITIALIZER;
```

Клиент/сервер: алгоритм работы сервера

1. Ожидание запроса
2. Поступил запрос - удалить запрос из очереди
3. Определить, какой запрос
 - чтение
 - запись
 - ВЫХОД
4. Сообщить клиенту о завершении, если клиент синхронный

Клиент/сервер: tty_server_routine(void *arg)

```
/* Сервер ожидает запросов, используя условную переменную
 * request. Сервер обрабатывает запросе в порядке FIFO. */
void *tty_server_routine (void *arg) {
    static pthread_mutex_t prompt_mutex =
                                                PTHREAD_MUTEX_INITIALIZER;

    request_t *request;
    int operation, len;
    for (;;) {
        pthread_mutex_lock (&tty_server.mutex);
        /* Ждать поступления данных */
        while (tty_server.first == NULL) {
            pthread_cond_wait(&tty_server.request,
                              &tty_server.mutex);
        }
        request = tty_server.first;
        tty_server.first = request->next;
        if (tty_server.first == NULL)
            tty_server.last = NULL;
        pthread_mutex_unlock(&tty_server.mutex);
    }
}
```

Клиент/сервер: tty_server_routine(void *arg)

```
/* Обработка данных */
operation = request->operation;
switch (operation) {
    case REQ_QUIT: break;
    case REQ_READ:
        if (strlen (request->prompt) > 0)
            printf (request->prompt);
        if (fgets (request->text, 128, stdin) == NULL)
            request->text[0] = '\0';
        /* Заменяем символ '\n' на '\0' */
        len = strlen (request->text);
        if (len > 0 && request->text[len-1] == '\n')
            request->text[len-1] = '\0';
        break;
    case REQ_WRITE:
        puts (request->text);
        break;
    default: break;
}
```

Клиент/сервер: tty_server_routine(void *arg)

```
if (request->synchronous) {
    pthread_mutex_lock (&tty_server.mutex);
    request->done_flag = 1;
    pthread_cond_signal (&request->done);
    pthread_mutex_unlock (&tty_server.mutex);
} else
    free(request);
if (operation == REQ_QUIT)
    break;
}
return NULL;
}
```

Клиент/сервер: отправка запроса серверу

1. Если сервер ещё не запущен, запустить его (detached).
2. Создать и проинициализировать структуру для пакета запроса.
3. Добавить запрос в очередь запросов.
4. Разбудить сервер, сообщив, что пакет доступен.
5. Если запрос синхронный, дождаться выполнения операции.

Клиент/сервер: tty_server_request(int operation, int sync, ...

```
/* Запрос на выполнение операции */  
void tty_server_request(int operation, int sync,  
                        const char *prompt, char *string) {  
    request_t *request;  
    int status;  
  
    pthread_mutex_lock(&tty_server.mutex);  
    if (!tty_server.running) {  
        pthread_t thread;  
        pthread_attr_t detached_attr;  
        pthread_attr_init (&detached_attr);  
        pthread_attr_setdetachstate(&detached_attr,  
                                     PTHREAD_CREATE_DETACHED);  
  
        tty_server.running = 1;  
        pthread_create(&thread, &detached_attr,  
                      tty_server_routine, NULL);  
        pthread_attr_destroy (&detached_attr);  
    }  
}
```

Клиент/сервер: `tty_server_request(int operation, int sync, ...`

```
/* Создать и проинициализировать структуру запроса */
request = (request_t *) malloc (sizeof(request_t));
request->next = NULL;
request->operation = operation;
request->synchronous = sync;

if (sync) {
    request->done_flag = 0;
    pthread_cond_init (&request->done, NULL);
}

if (prompt != NULL)
    strncpy (request->prompt, prompt, 32);
else
    request->prompt[0] = '\\0';

if (operation == REQ_WRITE && string != NULL)
    strncpy (request->text, string, 128);
else
    request->text[0] = '\\0';
```

Клиент/сервер: tty_server_request(int operation, int sync, ...

```
/* Сообщить серверу, что запрос доступен для обработки */
pthread_cond_signal (&tty_server.request);

/* Если запрос синхронный, дождаться ответа */
if (sync) {
    while (!request->done_flag)
        pthread_cond_wait(&request->done, &tty_server.mutex);

    if (operation == REQ_READ) {
        if (strlen (request->text) > 0)
            strcpy (string, request->text);
        else
            string[0] = '\0';
    }

    pthread_cond_destroy(&request->done);
    free (request);
}

pthread_mutex_unlock(&tty_server.mutex);
}
```

Клиент/сервер: клиент

1. Отправить запрос на чтение строки.
2. Если строка пустая, завершить выполнение
 - уменьшить число клиентов
 - разбудить главный поток, если это был последний клиент
3. Если не пустая, отправить несколько запросов на вывод прочитанной строки.

Клиент/сервер: client_routine (void *arg)

```
/* Процедура клиентов, запрашивающих сервер */
void *client_routine (void *arg) {
    int my_number = (int) arg, loops;
    char prompt[32], string[128], formatted[128];
    sprintf (prompt, "Client %d> ", my_number);
    for (;;) {
        tty_server_request(REQ_READ, 1, prompt, string);
        if (strlen(string) == 0) break;
        for (loops = 0; loops < 4; loops++) {
            sprintf(formatted, "(%d#%d) %s", my_number, loops, string);
            tty_server_request(REQ_WRITE, 0, NULL, formatted);
            sleep (1);
        }
    }
    pthread_mutex_lock (&client_mutex);
    if (--client_threads <= 0)
        pthread_cond_signal (&clients_done);
    pthread_mutex_unlock (&client_mutex);
    return NULL; }
```

Клиент/сервер: client_routine (void *arg)

```
int main (int argc, char *argv[]) {
    pthread_t thread;
    int count, status;

    /* Создать CLIENT_THREADS клиентов */
    client_threads = CLIENT_THREADS;
    for (count = 0; count < client_threads; count++) {
        pthread_create(&thread, NULL,
                      client_routine, (void *) count);
    }

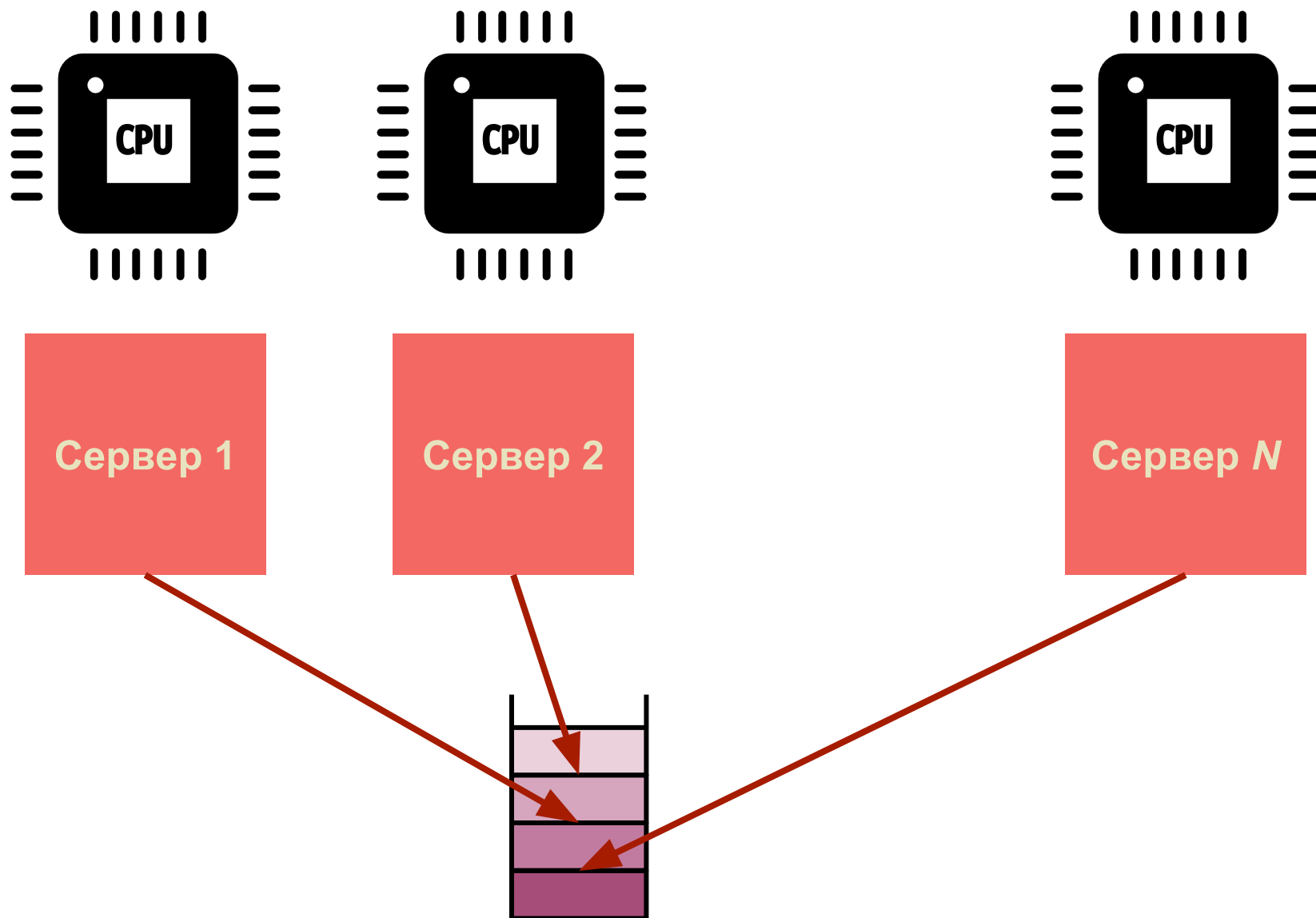
    pthread_mutex_lock (&client_mutex);
    while (client_threads > 0)
        pthread_cond_wait (&clients_done, &client_mutex);
    pthread_mutex_unlock (&client_mutex);

    printf( "All clients done\n");
    tty_server_request(REQ_QUIT, 1, NULL, NULL);
    return 0;
}
```

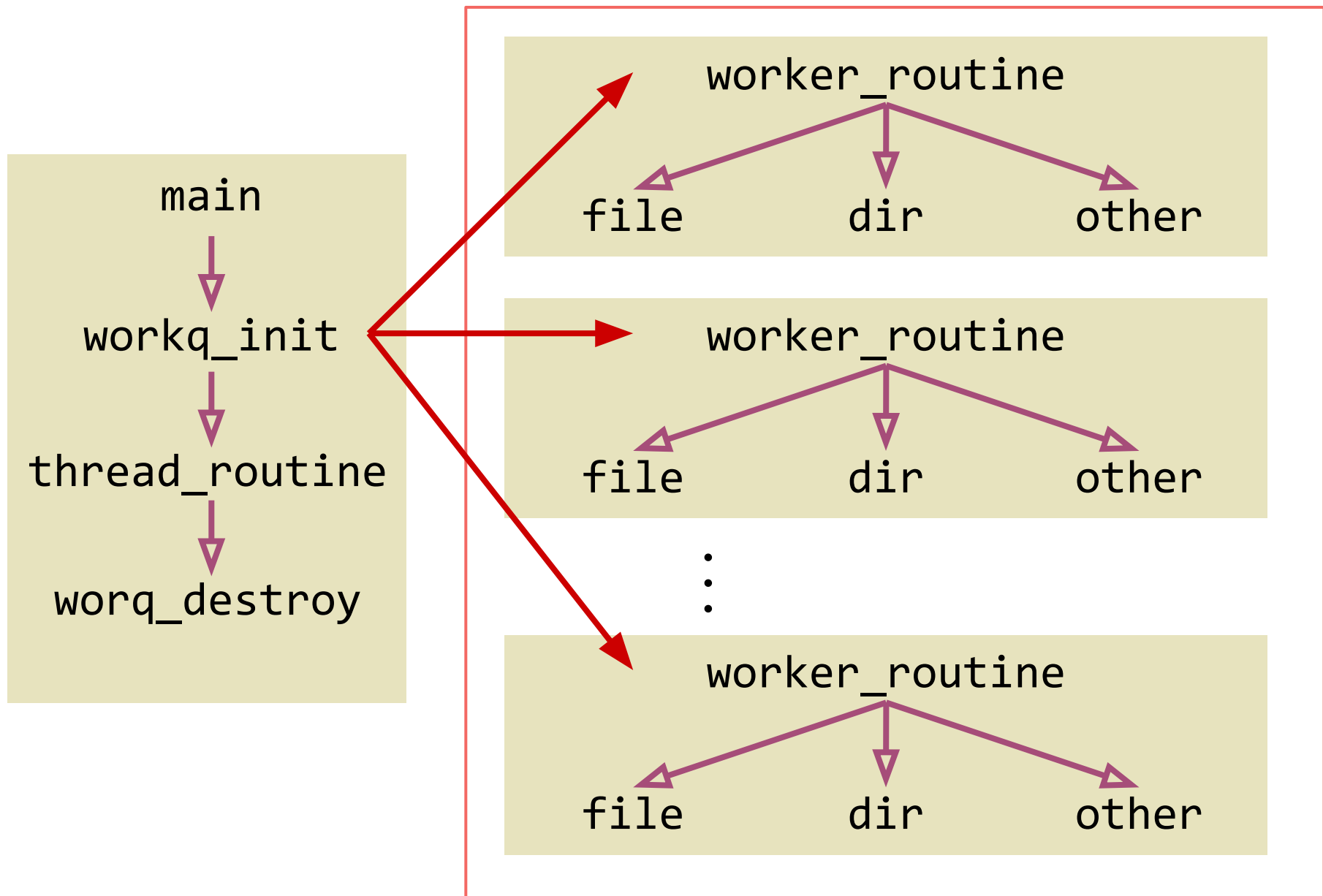
Очередь заданий (work queue)

- Набор потоков получает запросы из общей очереди, затем обрабатывает их параллельно.
- Менеджер очереди заданий можно рассмотреть как менеджера группы потоков (work crew).
- Задаётся максимальный уровень параллелизма.
- Потоки запускаются и останавливаются при необходимости выполнить задания.
- Потоки, для которых нет задания, ждут какое-то время и затем завершаются.
- Время такого ожидания определяется временем создания потока и стоимостью работы потока, не выполняющего полезной работы.

Очередь заданий (work queue)



Очередь заданий (work queue)



Очередь заданий: структуры

1. Элемент очереди заданий.
2. Сама очередь
 - защита
 - синхронизация (ожидание задания)
 - очередь заданий
 - параметры, счётчики
 - параллелизм
 - функция, выполняющая данное задание

Очередь заданий: workq.h

```
/* Структура для элемента (запроса) очереди заданий. */
typedef struct workq_ele_tag {
    struct workq_ele_tag *next;
    void *data;
} workq_ele_t;

/* Структура, описывающая очередь заданий. */
typedef struct workq_tag {
    pthread_mutex_t mutex;      /* контроль доступа к очереди */
    pthread_cond_t cv;          /* условие для ожидания задания */
    pthread_attr_t attr;        /* атрибут “отсоединённого” потока */
    workq_ele_t *first, *last;  /* очередь заданий */
    int valid;
    int quit;                   /* условие выхода */
    int parallelism;             /* максимум потоков */
    int counter;                 /* текущее число потоков */
    int idle;                    /* число простаивающих потоков */
    void (*engine)(void *arg); /* поток, выполняющий задания */
} workq_t;

#define WORKQ_VALID 0x0ct2014
```

Очередь заданий (work queue): пример

```
/* Интерфейс для создания и удаления очереди заданий, а также  
 * функция для добавления заданий в очередь */
```

```
extern int workq_init (  
    workq_t *wq,                /* максимум потоков */  
    int threads,  
    void (*engine)(void *));    /* функция для выполнения  
                                * заданий */
```

```
extern int workq_destroy (workq_t *wq);
```

```
extern int workq_add (workq_t *wq, void *data);
```


Очередь заданий: workq_init

```
/* Инициализировать очередь заданий. */
int workq_init (workq_t *wq, int threads,
               void (*engine)(void *arg)) {
    pthread_attr_init(&wq->attr);
    pthread_attr_setdetachstate(&wq->attr,
                               PTHREAD_CREATE_DETACHED);
    pthread_mutex_init(&wq->mutex, NULL);
    pthread_cond_init(&wq->cv, NULL);

    wq->quit = 0; /* выход не сейчас */
    wq->first = wq->last = NULL; /* очередь пуста */
    wq->parallelism = threads; /* макс. число серверов */
    wq->counter = 0; /* пока нет серверов */
    wq->idle = 0; /* нет простаивающих */
    wq->engine = engine; /* поток, выполняющий
                          * работу */

    wq->valid = WORKQ_VALID;
    return 0;
}
```

Очередь заданий: уничтожение очереди заданий

1. Проверить статус, запретить поступление новых заданий.
2. Если ещё есть активные потоки, завершить их:
 - установить флаг выхода
 - если есть простаивающие потоки, разбудить их
3. Подождать, пока все потоки не будут завершены.

Очередь заданий: workq_destroy

```
/* Уничтожить очередь заданий */
int workq_destroy (workq_t *wq) {
    int status, status1, status2;
    pthread_mutex_lock (&wq->mutex);
    wq->valid = 0;      /* предотвращать другие операции */

    /* Если активны другие потоки, остановить их
     * 1. установить флаг выхода,
     * 2. разбудить все потоки,
     * 3. ожидать завершения работы всех потоков */
    if (wq->counter > 0) {    /* если нет серверов */
        wq->quit = 1;
        /* если какие-то потоки простаивают, разбудить их */
        if (wq->idle > 0)
            pthread_cond_broadcast(&wq->cv);
    }
}
```

Очередь заданий: workq_destroy

```
/* Условие cv используется для двух отдельных предикатов.  
 * Это нормально, поскольку нижеприведённый случай  
 * срабатывает только 1 раз - при завершении потока. */
```

```
/* Разбудили и теперь ждём завершения серверов... */
```

```
while (wq->counter > 0) {  
    pthread_cond_wait(&wq->cv, &wq->mutex);  
}
```

```
pthread_mutex_unlock(&wq->mutex);  
pthread_mutex_destroy(&wq->mutex);  
pthread_cond_destroy(&wq->cv);  
pthread_attr_destroy(&wq->attr);
```

```
}
```

Очередь заданий: добавление задания

1. Создание и инициализация структуры для нового задания.
2. Добавление нового задания в очередь заданий.
3. Если есть простаивающие рабочие потоки (которым нечего делать), разбудить один.
4. Если нет спящих рабочих потоков, создать один, если есть куда создавать.
5. Если создавать больше некуда, выйти из функции, оставив новое задание в очереди.

Очередь заданий: workq_add

```
/* Добавить задание в очередь */
int workq_add (workq_t *wq, void *element) {
    workq_ele_t *item;
    pthread_t id;
    int status;
    if (wq->valid != WORKQ_VALID)
        return EINVAL;

    /* Создать и проинициализировать структуру. */
    item = (workq_ele_t *)malloc (sizeof(workq_ele_t));
    item->data = element;
    item->next = NULL;
    pthread_mutex_lock(&wq->mutex);

    /* Добавить запрос в конец очереди, обновить указатели */
    if (wq->first == NULL)
        wq->first = item;
    else
        wq->last->next = item;
    wq->last = item;
}
```

Очередь заданий: workq_add

```
/* Если потоки бездействуют, разбудить один */
if (wq->idle > 0) {
    pthread_cond_signal(&wq->cv);
} else if (wq->counter < wq->parallelism) {

    /* Если нет покоящихся потоков и можно создать один,
       * создаём один, как раз под задачу */
    printf("Creating new worker\n");
    pthread_create(&id, &wq->attr, workq_server, (void*) wq);
    wq->counter++;
}

pthread_mutex_unlock(&wq->mutex);
return 0;
}
```

Очередь заданий: сервер

1. Если заданий нет и не надо выходить, ждём задания какое-то заданное время.
2. После окончания ожидания (не важно по какой причине):
 - если есть задание в очереди, выполняем его
 - если поступил запрос на завершение, будим все рабочие потоки и выходим
 - если истёк таймер, завершаем работу сервера

Очередь заданий: workq_server

```
/* Поток запускается и начинает обслуживать очередь. */  
  
static void *workq_server(void *arg) {  
    struct timespec timeout;  
    workq_t *wq = (workq_t *) arg;  
    workq_ele_t *we;  
    int timedout;  
    printf("A worker is starting\n");  
  
    /* Сначала создаётся очередь, потом потоки;  
     * сначала уничтожаются потоки, потом очередь. */  
  
    pthread_mutex_lock (&wq->mutex);  
    for (;;) {  
        timedout = 0;  
        printf("Worker waiting for work\n");  
        clock_gettime (CLOCK_REALTIME, &timeout);  
        timeout.tv sec += 2;
```

Очередь заданий: workq_server

```
while (wq->first == NULL && !wq->quit) {  
    /* Сервер ждёт нового задания 2 секунды,  
     * потом завершает работу. */  
    status = pthread_cond_timedwait(&wq->cv, &wq->mutex,  
                                    &timeout);  
  
    if (status == ETIMEDOUT) {  
        printf("Worker wait timed out\n");  
        timeout = 1;  
        break;  
    } else if (status != 0) {  
  
        /* Событие маловероятно. Для простоты здесь сервер  
         * завершается и задание подхватывает другой сервер */  
        wq->counter--;  
        pthread_mutex_unlock(&wq->mutex);  
        return NULL;  
    }  
}
```

Очередь заданий: workq_server

```
printf("Work queue: %#1x, quit: %d\n",  
      wq->first, wq->quit));  
we = wq->first;  
if (we != NULL) {  
    wq->first = we->next;  
    if (wq->last == we)  
        wq->last = NULL;  
    pthread_mutex_unlock (&wq->mutex);  
    printf("Worker calling engine\n");  
    wq->engine (we->data);  
    free (we);  
    pthread_mutex_lock (&wq->mutex);  
}
```

Очередь заданий: workq_server

```
/* Если нет больше заданий, и поступил запрос
 * на завершение работы серверов, завершить работу. */
if (wq->first == NULL && wq->quit) {
    printf("Worker shutting down\n");
    wq->counter--;
    if (wq->counter == 0) pthread_cond_broadcast(&wq->cv);
    pthread_mutex_unlock(&wq->mutex);
    return NULL;
}
/* Если нет больше заданий, и мы ожидаем так долго,
 * как можно, завершить работу этого потока сервера. */
if (wq->first == NULL && timeout) {
    printf("engine terminating due to timeout.\n");
    wq->counter--;
    break;
}
}
pthread_mutex_unlock(&wq->mutex);
printf("worker exiting\n"); return NULL; }
```

Очередь заданий: `workq_main.c`

Пример

Два потока параллельно отправляют запросы на выполнение задание (возведение числа в степень). Реализуется сбор статистики.

Очередь заданий - пример: структуры

1. Задание: данные (число и степень), поток и статистика (число вызовов).
2. Локальные данные потоков для сбора статистики.
3. Мьютексы.
4. Список заданий.

Очередь заданий: workq_main.c

```
#include "workq.h"

#define ITERATIONS 25

typedef struct power_tag {
    int value;
    int power;
} power_t;

typedef struct engine_tag {
    struct engine_tag *link;
    pthread_t thread_id;
    int calls;
} engine_t;

/* Для отслеживания числа активных серверов */
pthread_key_t engine_key;

pthread_mutex_t engine_list_mutex = PTHREAD_MUTEX_INITIALIZER;

engine_t *engine_list_head = NULL;

workq_t workq;
```

Очередь заданий: workq_main.c

```
/* Деструктор локальных данных потоков. */  
void destructor (void *value_ptr)  
    engine_t *engine = (engine_t *) value_ptr;  
    pthread_mutex_lock (&engine_list_mutex);  
    engine->link = engine_list_head;  
    engine_list_head = engine;  
    pthread_mutex_unlock (&engine_list_mutex);  
}
```


Очередь заданий: workq_main.c

```
/* Функция, вызываемая серверами очередей для
 * выполнения задания */
void engine_routine (void *arg) {
    engine_t *engine;
    power_t *power = (power_t *) arg;
    int result, count;
    engine = pthread_getspecific(engine_key); // сбор статистики
    if (engine == NULL) {
        engine = (engine_t*) malloc (sizeof(engine_t));
        pthread_setspecific(engine_key, (void*) engine);
        engine->thread_id = pthread_self ();
        engine->calls = 1;
    } else
        engine->calls++;
    result = 1;
    printf("Engine: computing %d^%d\n", power->value, power->power);
    for (count = 1; count <= power->power; count++)
        result *= power->value; // вычисление степени
    free(arg); }
```

Очередь заданий: workq_main.c

```
/* Поток, генерирующий задания. */
void *thread_routine(void *arg) {
    power_t *element;
    int count;
    unsigned int seed = (unsigned int) time(NULL);
    int status;

    /* Генерация запросов */
    for (count = 0; count < ITERATIONS; count++) {
        element = (power_t*) malloc(sizeof (power_t));
        element->value = rand_r (&seed) % 20;
        element->power = rand_r (&seed) % 7;
        printf("Request: %d^%d\n ", element->value, element->power);
        status = workq_add (&workq, (void*) element);
        if (status != 0)
            err_abort("Add to work queue");
        sleep (rand_r (&seed) % 5);
    }
    return NULL;
}
```

Очередь заданий: workq_main.c

```
int main (int argc, char *argv[]) {
    pthread_t thread_id;
    engine_t *engine;
    int count = 0, calls = 0, status;
    pthread_key_create (&engine_key, destructor);
    workq_init (&workq, 4, engine_routine);
    pthread_create (&thread_id, NULL, thread_routine, NULL);
    (void) t hread_routine (NULL); /* второй поток */
    pthread_join (thread_id, NULL);
    workq_destroy (&workq);
    engine = engine_list_head;

    while (engine != NULL) { /* Подсчитать статистику */
        count++;
        calls += engine->calls;
        printf ("engine %d: %d calls\n", count, engine->calls);
        engine = engine->link;
    }
    printf ("%d engine threads processed %d calls\n", count, calls);
    return 0; }
```