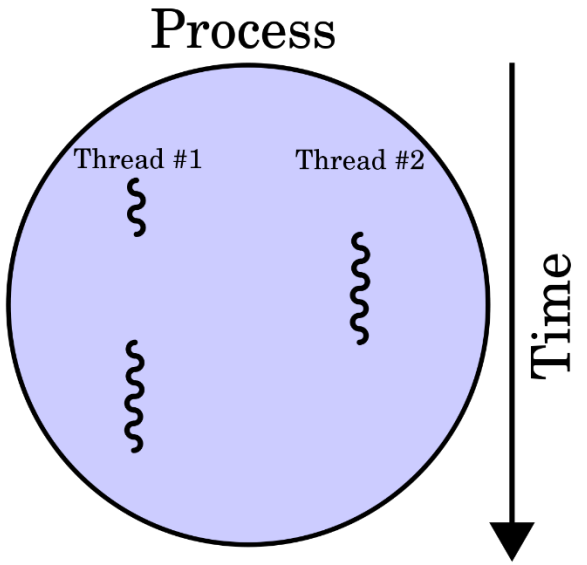
Курс «Проектирование больших систем на языке C++»

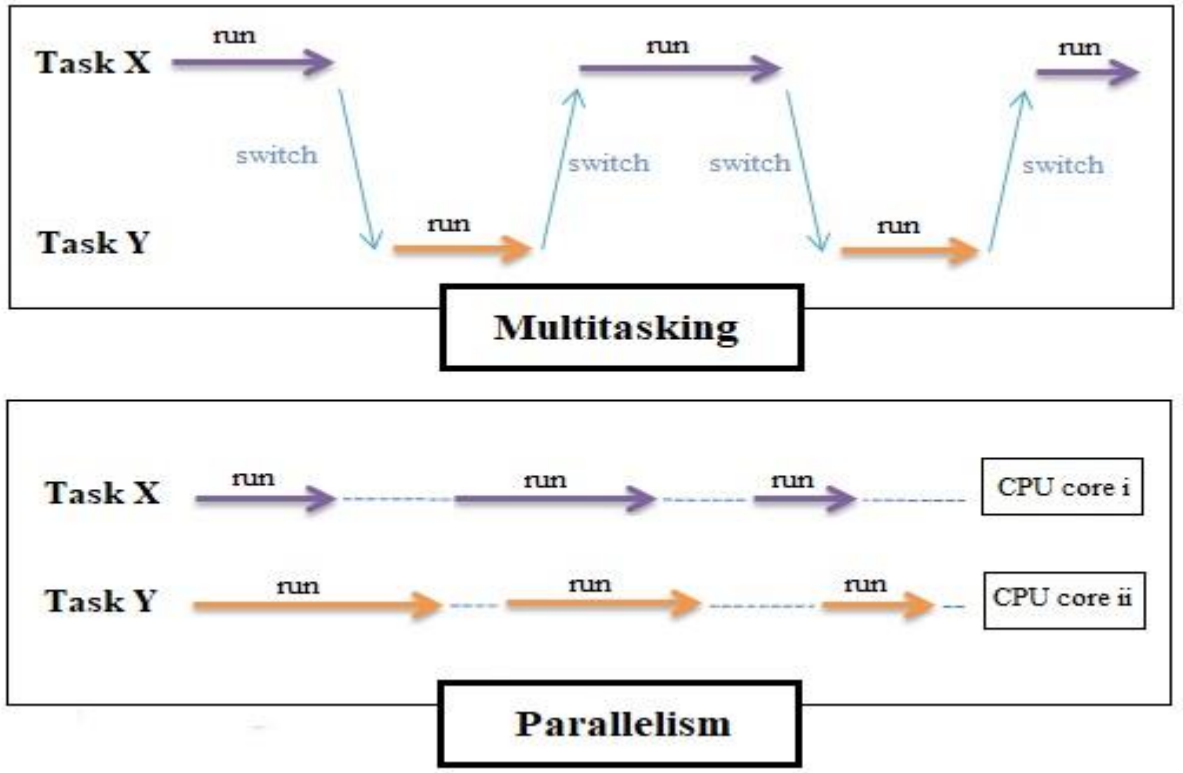
Лекция

Процессы и Потоки

Процесс исполнения (Process)

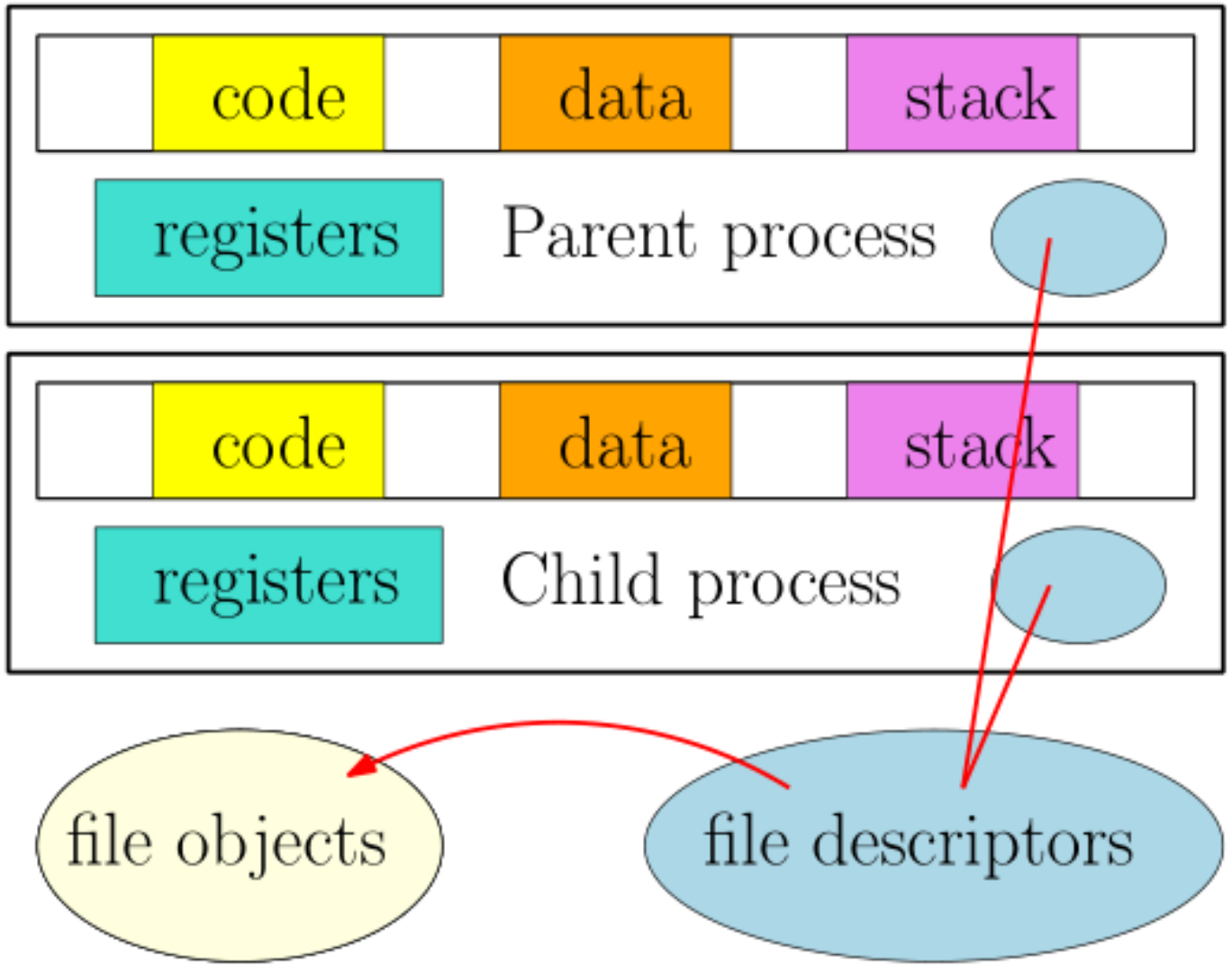
Процесс





- **Программа** – написанный на некотором формализованном языке алгоритм, решающий поставленную задачу.
- **Процесс** – динамическая сущность программы, её код в процессе своего выполнения, имеет: собственную область памяти под код и данные, состояние.
- **Поток выполнения** (нить) – часть процесса – последовательность выполняемых процессором инструкций.

Создание процессов

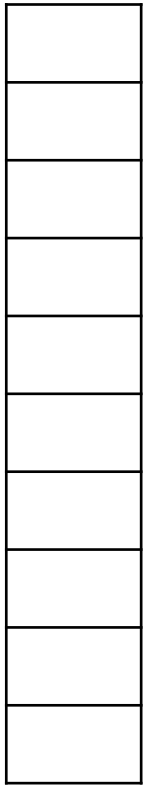


Создание процессов

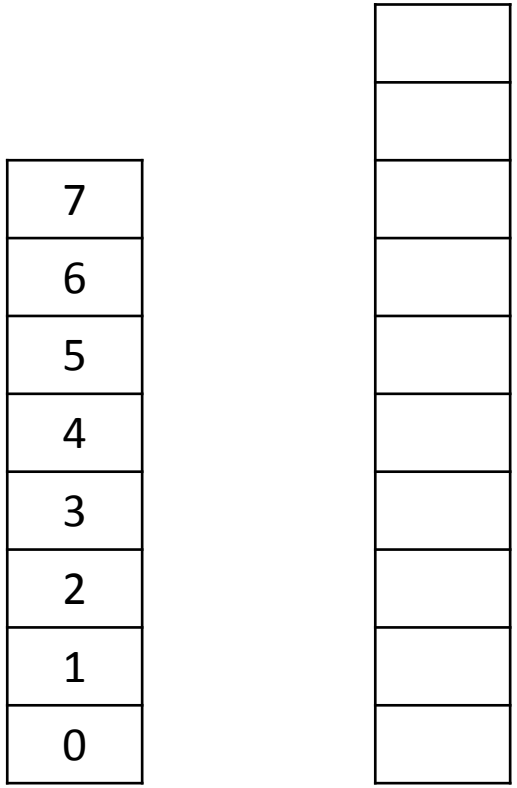
```
1.  if (!fork()) {
2.      // child
3.      write(1, "hello ", 6);
4.      exit(0);
5.  } else {
6.      // parent
7.      waitpid(-1, 0, 0); //нет гонки процессов
8.      write(1, "world\n", 6);
9.  }
```

Вывод: hello world

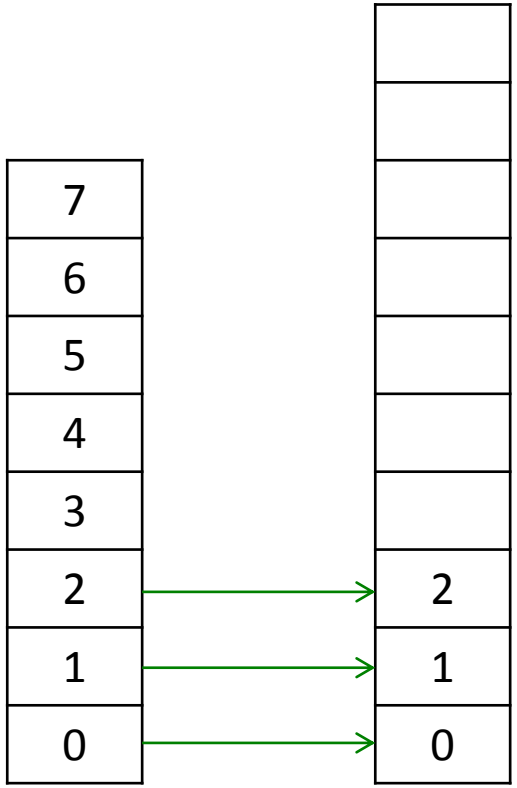
Copy-on-write



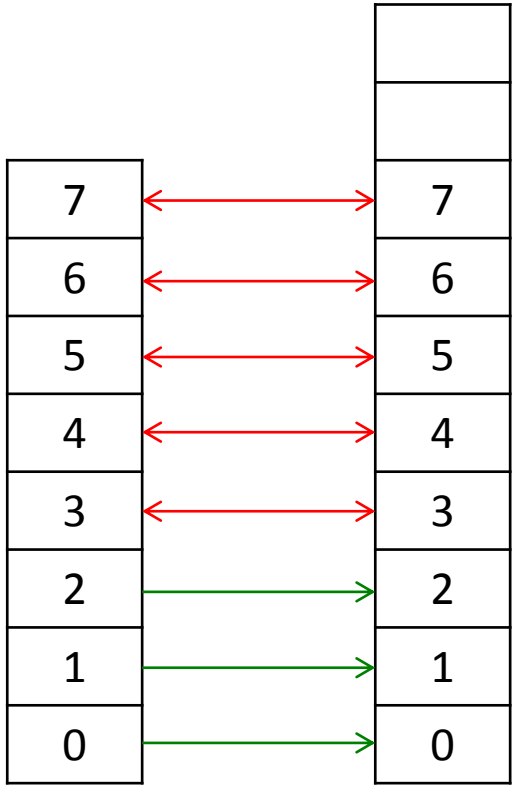
Copy-on-write



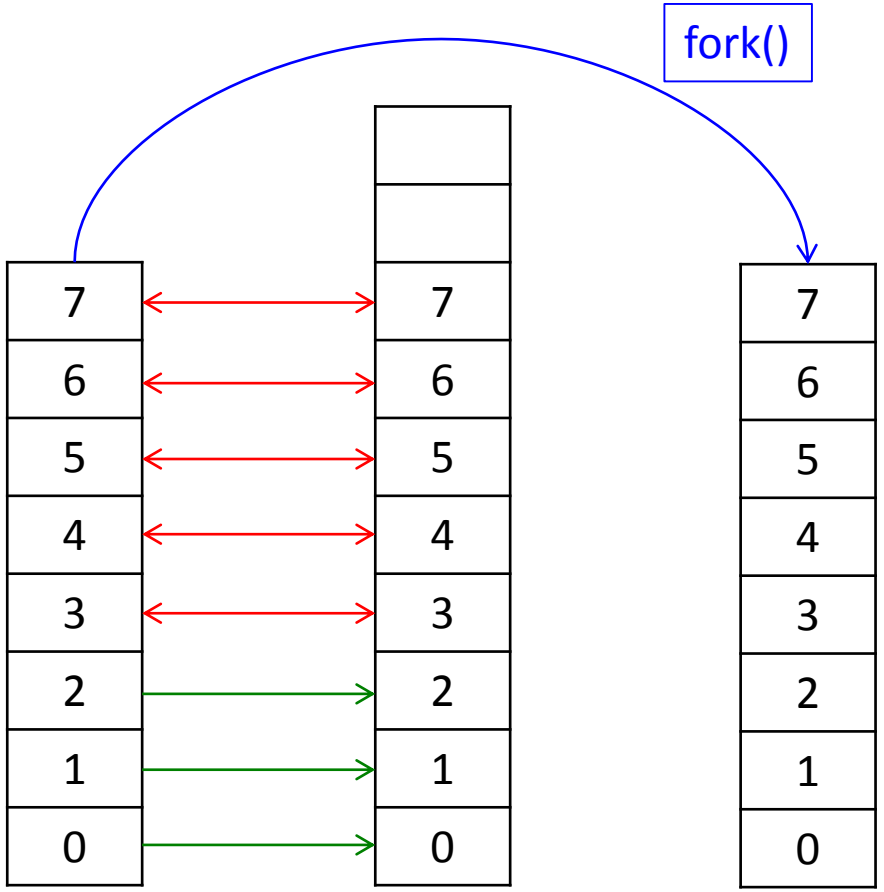
Copy-on-write



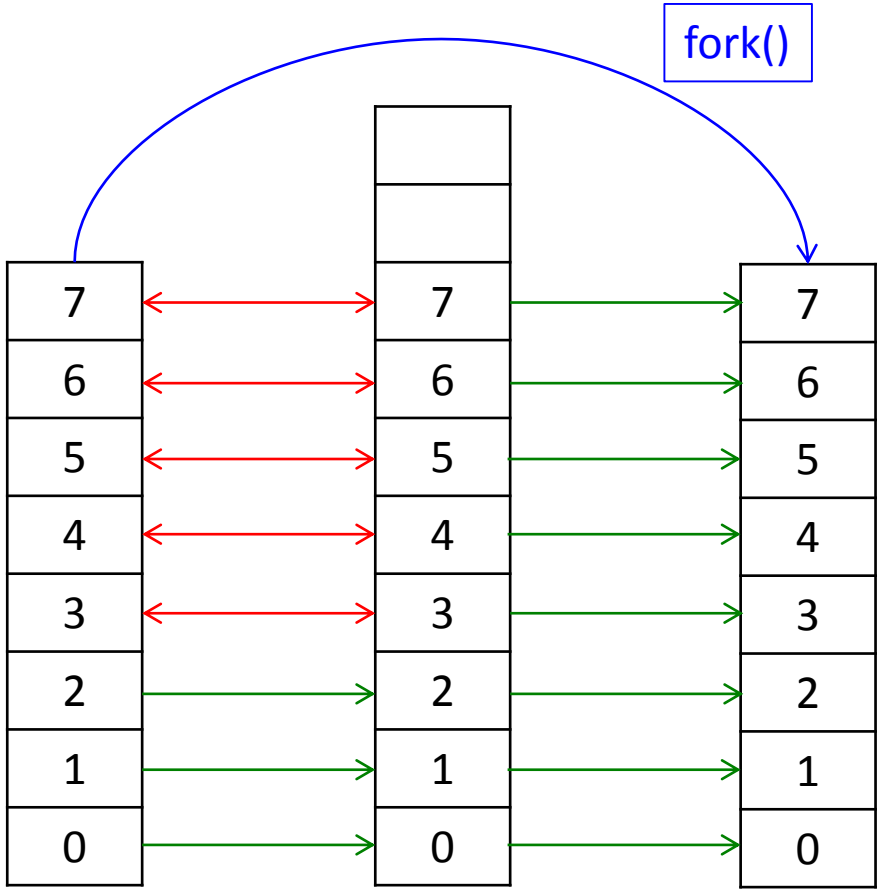
Copy-on-write



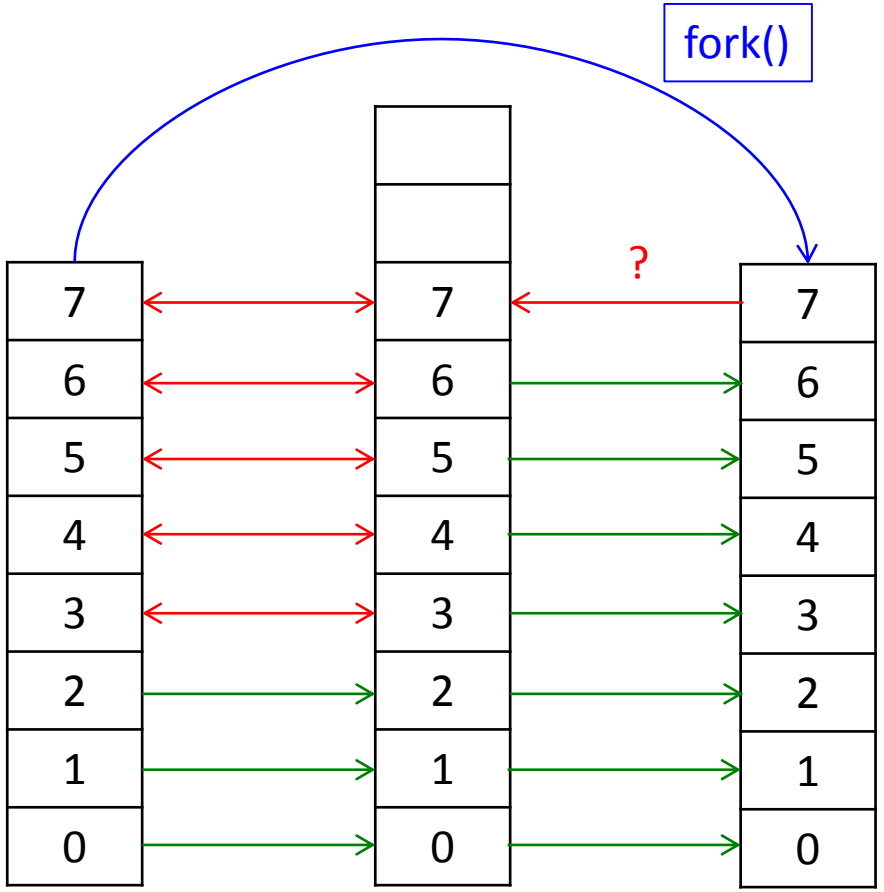
Copy-on-write



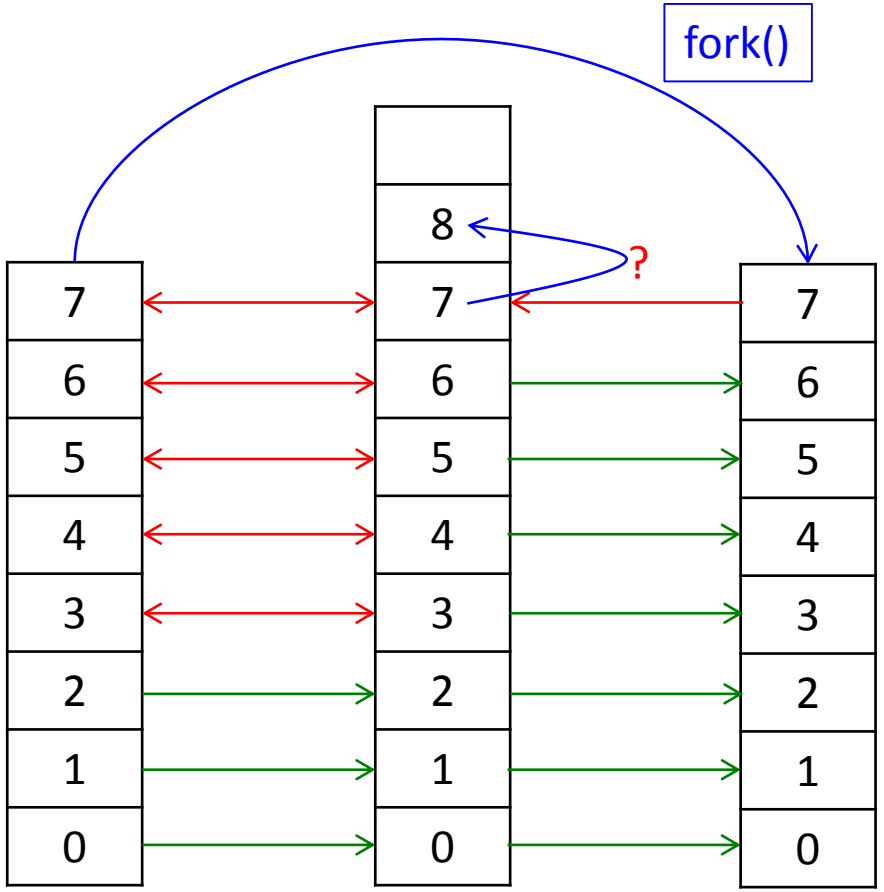
Copy-on-write



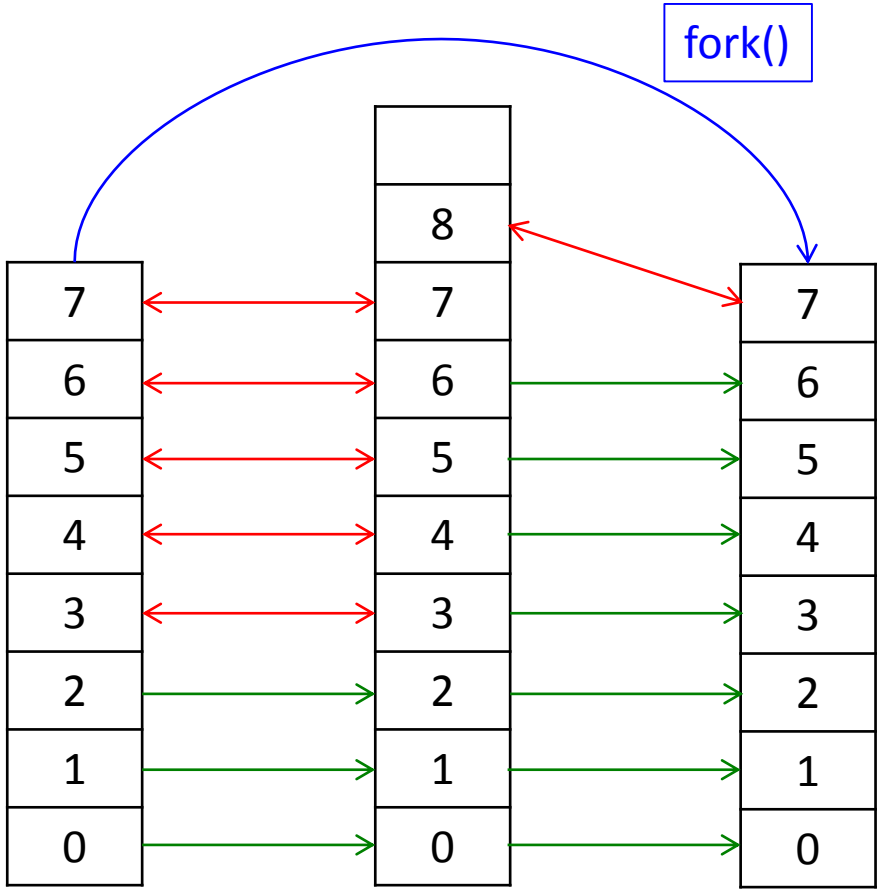
Copy-on-write



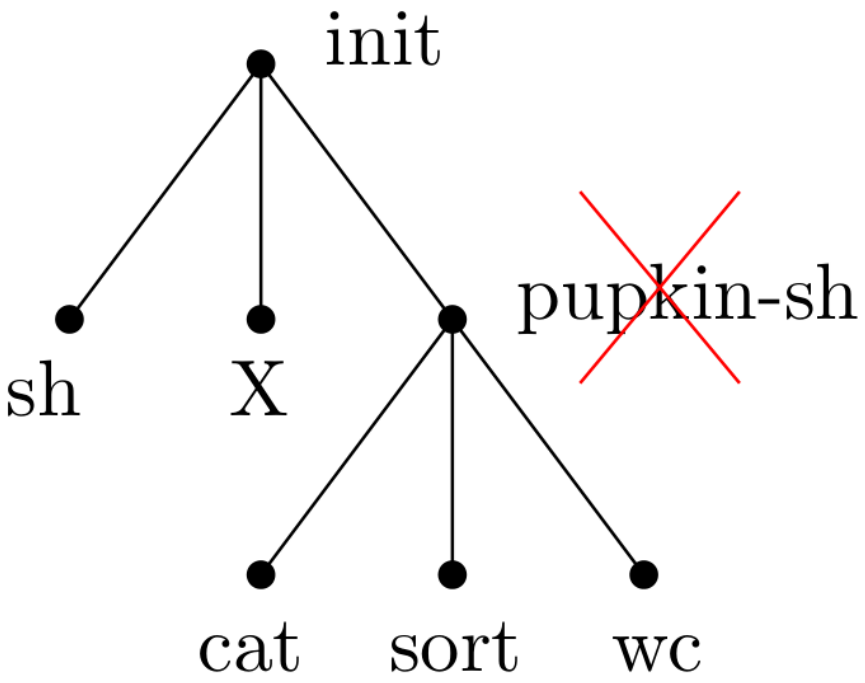
Copy-on-write

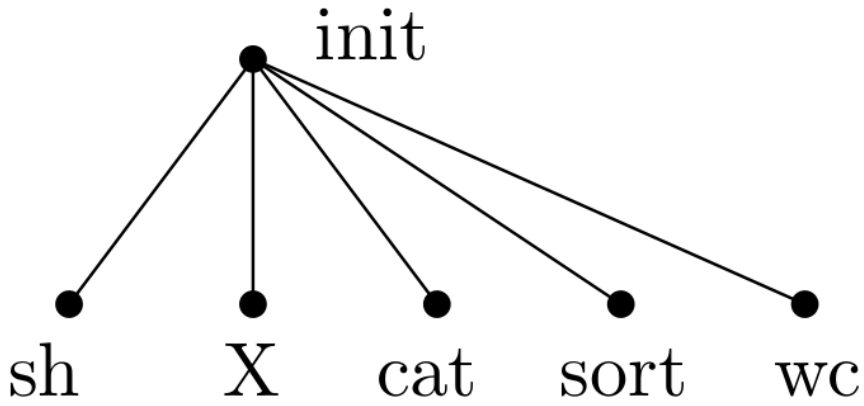


Copy-on-write



Иерархия процессов





PCB
Process Control Block

Блок информации о процессе:

- **PID** - ID процесса;
- **PPID** - ID процесса-родителя;
- **UID**, **EUID**, **GID**, **EGID**,
- путь и аргументы, с которым запущен процесс;
- программный счётчик;
- указатели стэка;

и др.

Атрибуты процесса

UID, GID, EUID, EGID

```
1.   pid_t getpid(); // PID=1 для init.
2.  pid_t getppid(); // Если родительский процесс завершается, потомок получает PPID=1.

3.   // Кто создал? Реальные идентификаторы пользователя и группы
4.   uid_t getuid();
5.   int setuid(uid_t uid);
6.   gid_t getgid();
7.   int setgid(gid_t gid);

8.   // “От чьего лица выполняется?”
9.   // (Эффективные идентификаторы пользователя и группы).
10.  uid_t geteuid();
11.  int seteuid(uid_t uid);
12.  gid_t getegid();
13.  int setegid(gid_t gid);
```

Атрибуты процесса

Приоритет

```
1.    int nice(int incr);

2.    // NZERO = 20
3.    // NICE = 0..39
4.    // NICE-20 legacy
5.    // -1 == error или приоритет
```

Атрибуты процесса

Ограничения

```
1.    long ulimit(int cmd, …); //deprecated

2.    int getrlimit(int resource, struct rlimit *rlp);
3.    int setrlimit(int resource, const struct rlimit *rlp);
```

```
RLIMIT_CORE    - размер coredump,
RLIMIT_CPU     - реальное время процесса между вытеснениями (cpu_time – sec.),
RLIMIT_DATA    - размер сегмента данных,
RLIMIT_FSIZE   - размер файла,
RLIMIT_NOFILE  - количество открытых файлов,
RLIMIT_STACK   - размер стэка,
RLIMIT_AS      - размер, сколько всего памяти
```

```
1.    struct rlimit {
2.       rlim_t rlim_cur;
3.       rlim_t rlim_max;
4.    }
```

```
RLIM_SAVED_MAX – устанавливается root’ом,
RLIM_SAVED_CUR – устанавливаем мы, можно=RLIM_INFINITY<=MAX
```

Атрибуты процесса

Переменные окружения

```
1.  // Имя=Значение
2.  // Имя=Значение
3.  // ...
4.  // Имя=Значение
5.  // \0
6.  extern char **environ;

7.  char* getenv(const char *var);
8.  int putenv(char *str);
9.  int setenv(const char *var, const char *val, int overwrite);
10. int unsetenv(const char *var);
```

exit code

- По завершению процесс возвращает целочисленный **код возврата программы**
- обычно 0 – успех, остальные - ошибка

Процесс может завершиться

- **нормально**:
  вызов exit или return из main
- **ошибочно**: критическая
  SegFault, GenProtectException, DivByZero, …
- **принудительно**:
  не обработан какой-то из посланных сигналов

Порождение процессов

Порождение процесса через exec

- `int execl(const char *path, const char *arg, ...);`
- `int execv(const char *path, char *const argv[]);`
- `int execle(const char *path, const char *arg, ..., char * const envp[]);`
- `int execve(const char *path, char *const argv[], char *const envp[]);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execvp(const char *file, char *const argv[]);`

fork + exec

```
1.  if (!fork()) {
2.     // child
3.     char *argv[] = {
4.         "/bin/echo",
5.         "-n", "hello ", 0};
6.     char *envp[] = {0};
7.     execve("/bin/echo", argv, envp);
8.  } else {
9.     // parent
10.    waitpid(-1, 0, 0);
11.    write(1, "world\n", 6);
12. }
```

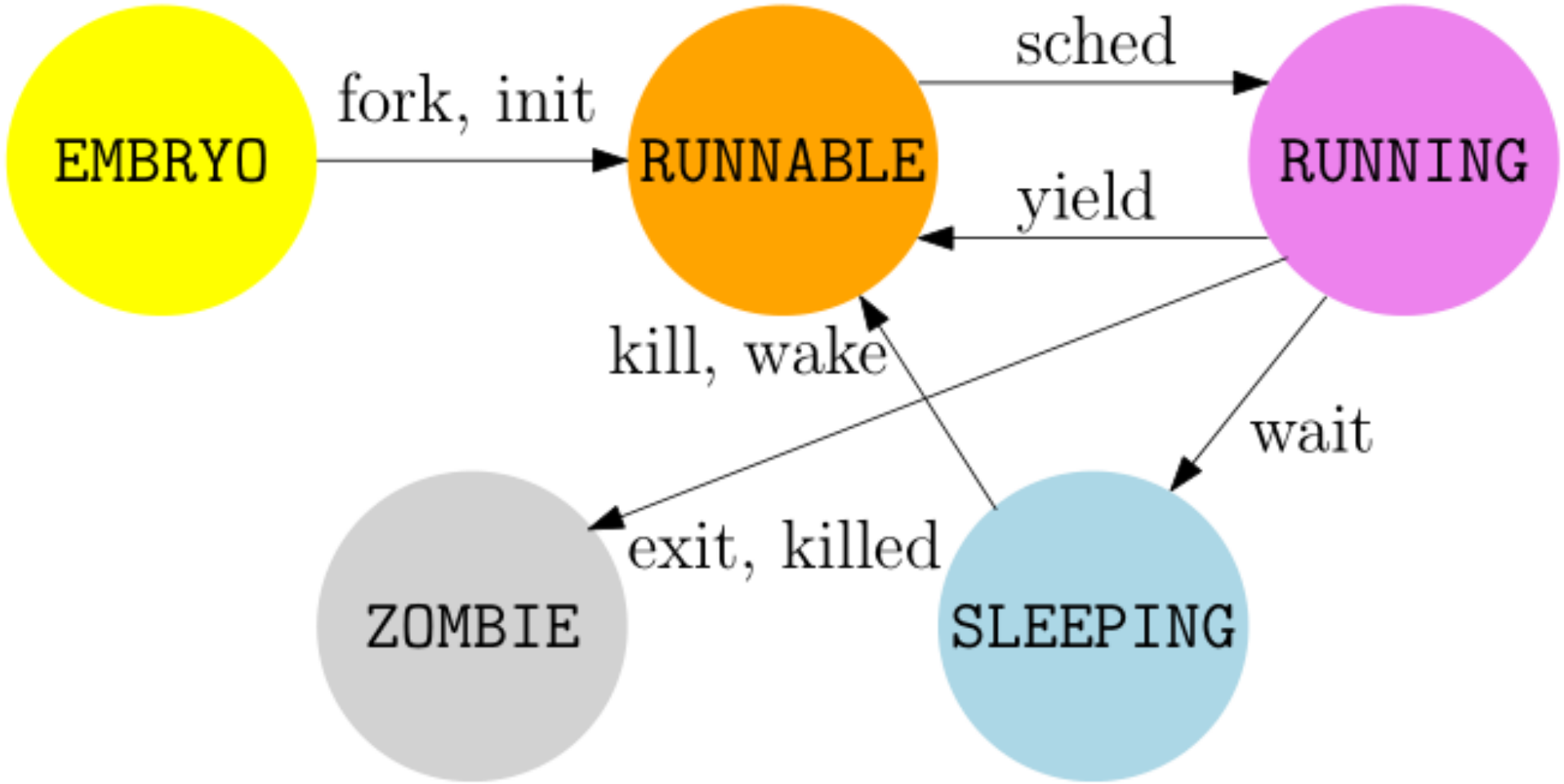
Вывод: hello world

Порождение процессов

Порождение процесса через system

```
1.    int system(const char *command);
```

Жизненный цикл процесса



Завершение процессов

Как предотвратить зомби?

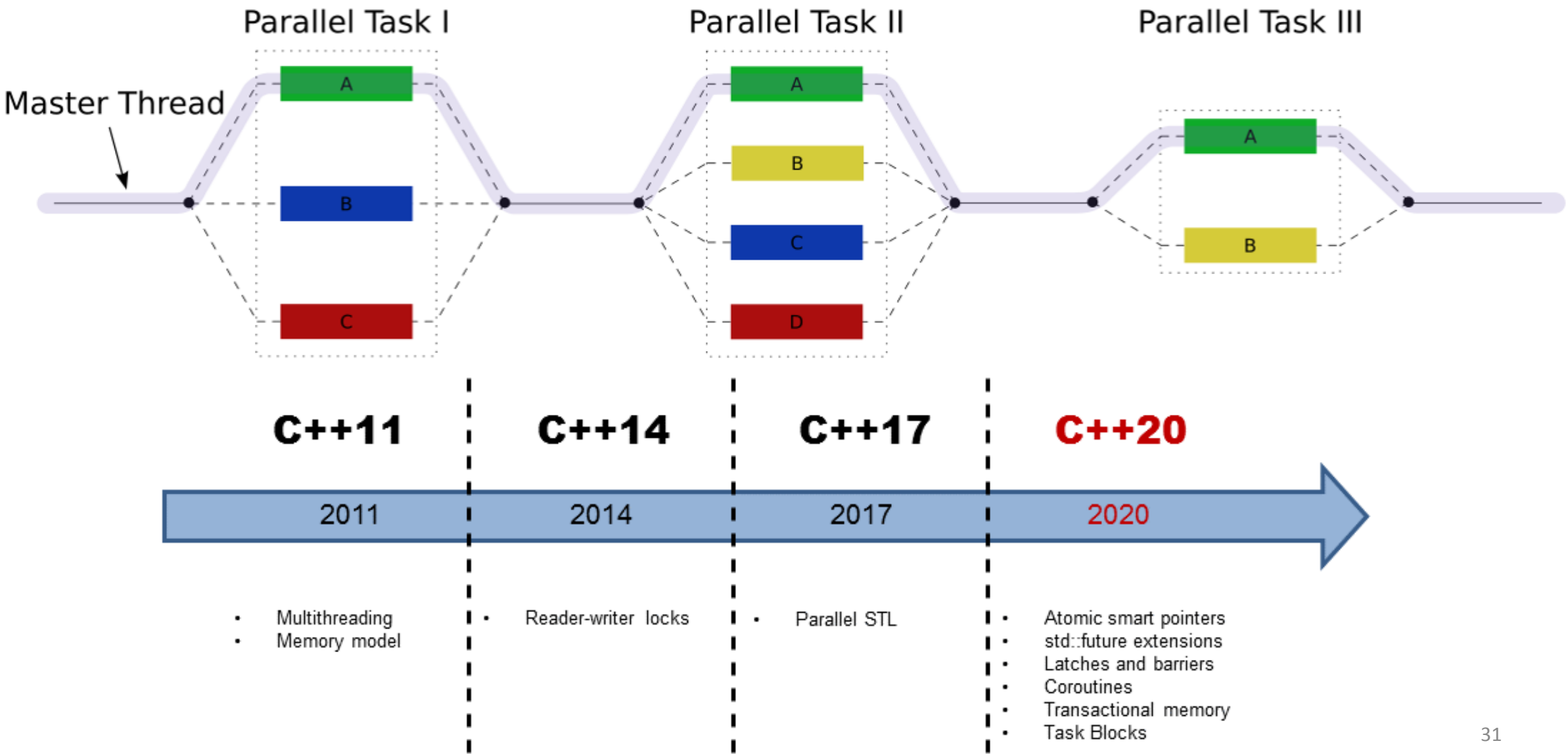
```
1.  pid_t waitpid(pid_t pid, int *statusp, int options);


2.  //ждём PID - конкретного или -1 – любого из потомков
3.  //pid <- вернётся, что умерло, или -1, если ничего
4.  //options <- WNOHANG – не блокирующий(просто проверка)
5.  //wait(&status) = waitpid(-1, &status, 0);
```

Завершение процесса
Способы

- exit() – процесс завершается сам. Процесс указывает код возврата, который будет доступен через wait
- kill() – процесс завершает другой процесс, отправляя сигнал

Поток исполнения (Thread)

Параллельная обработка



Создание потока

Создание потока на Си

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

2.   int pthread_create(
3.      pthread_t *thread,
4.      const pthread_attr_t *attr,
5.      void (*start_routine)(void*),
6.      void *arg);


7.   void* thread_func(void *value) {
8.      int *int_value = (int*)value;
9.      (*int_value)++;
10.     return value;
11.  }
```

Ожидание потока

Ожидание потока на Си

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

2.    //Ожидаем поток.
3.    int pthread_join(pthread_t thread, void **retval);

4.    //Отпускаем поток.
5.    int pthread_detach(pthread_t thread);
```

Атрибуты потока

Атрибуты потока на Си (detachstate)

```
1.   //Атрибуты задаются в момент создания потока.
2.   //Позже их изменить уже невозможно.
3.   pthread_attr_t * attr;

4.   int pthread_attr_init(pthread_attr_t *attr);
5.   int pthread_attr_destroy(pthread_attr_t *attr);

6.   int pthread_attr_getdetachstate(const pthread_attr_t *attr,
7.       int *detachstate);
8.   int pthread_attr_setdetachstate(pthread_attr_t *attr,
9.       int detachstate);

10. // PTHREAD_CREATE_DETACHED
11. // PTHREAD_CREATE_JOINABLE
```

Завершение потока

Неявное завершение потока (Implicit termination)

```
1.   void* thread_func(void *value) {
2.      int *int_value = (int*)value;
3.      (*int_value)++;
4.      return value;
5.   }
```

Завершение потока

Явное завершение потока (Explicit termination)

```
1.   //Внутри потока
2.   void pthread_exit(void *retval);

3.   //Из другого потока
4.   int pthread_cancel(pthread_t thread);

5.   //Понятие «Точка завершения»

6.   //read, waitpid, pthread_wait_condition
7.   //65 шт. всегда служат точками отмены
8.   //159 шт. могут выполнять эту роль

9.   //pthread_* не являются точками завершения
10.  //free, malloc, calloc, realloc тоже не точки завершения

11.  void pthread_testcancel();
```

Завершение потока

Ещё несколько очень полезных функций

```
1.  int pthread_setcancelstate(int state, int *oldstate);
2.  //PTHREAD_CANCEL_ENABLE
3.  //PTHREAD_CANCEL_DISABLE

4.  int pthread_setcanceltype(int type, int *oldtype);
5.  //PTHREAD_CANCEL_DEFERRED
6.  //PTHREAD_CANCEL_ASYNCHRONOUS

1.  void pthread_cleanup_push(void (*routine)(void*), void *arg);
2.  void pthread_cleanup_pop(int execute);

1.  int pthread_join(pthread_t thread, void **retval);
2.  // *retval = PTHREAD_CANCELED
```

C++11: Создание потока

Создание потока на C++11

```
1.   void ThreadFunction() {
2.       // ...
3.   }
4.   void ThreadFunction(int i, double d, std::string &s) {
5.       // ...
6.   }
7.   void ThreadFunction(int &a) {
8.       // ....
9.   }

10.  int main() {
11.      int b = 16;
12.      std::thread thread1(ThreadFuction);
13.      std::thread thread2(ThreadFuction, 16, 3.14, "Hello, world!");
14.      std::thread thread3(ThreadFuction, std::ref(b));
15.      // ...
16.  }
```

C++11: Ожидание потока

join и detach

```
1.  thread1.join();
2.  thread2.detach();
```

Мьютексы

Обычный мьютекс – создание и удаление, атрибуты

```
1.   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

```
1.   int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutex_attr_t *mattrp);
2.   int pthread_mutex_destroy(pthread_mutex_t *mp);
```

```
3.   int pthread_mutexattr_init(pthread_mutexattr_t *attr);
4.   int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Мьютексы

Обычный мьютекс – блокировки

```
1.    int pthread_mutex_lock(pthread_mutex_t *mp);
2.    int pthread_mutex_unlock(pthread_mutex_t *mp);
3.    int pthread_mutex_trylock(pthread_mutex_t *mp);
```

Мьютексы

Спин-блокировки

```
1.   int pthread_spin_init(pthread_spinlock_t *lock, int pshared);
2.   int pthread_spin_destroy(pthread_spinlock_t *lock);

3.   int pthread_spin_lock(pthread_spinlock_t *lock);
4.   int pthread_spin_trylock(pthread_spinlock_t *lock);
5.   int pthread_spin_unlock(pthread_spinlock_t *lock);
```

Мьютексы

RW-блокировки – создание и удаление, атрибуты

```
1.   int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);
2.   int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);

3.   pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
```

```
4.   int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
5.   int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Мьютексы

RW-блокировки – установка и снятие

```
1.    int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
2.    int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);

3.    int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
4.    int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);

5.    int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

Условные переменные

Создание и удаление

```
1.   int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
2.   int pthread_cond_destroy(pthread_cond_t *cond);

3.   pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Условные переменные

Ожидание и пробуждение

```
1.   int pthread_cond_wait(pthread_cond_t *cond,
2.                         pthread_mutex_t *mutex);
3.   int pthread_cond_timedwait(pthread_cond_t *restrict cond,
4.                              pthread_mutex_t *restrict mutex,
5.                              const struct timespec *restrict abstime);

6.   int pthread_cond_signal(pthread_cond_t *cond);
7.   int pthread_cond_broadcast(pthread_cond_t *cond);
```

Условные переменные

Пример

```
1.   void *consumer(void *args) {
2.      while (1) {
3.          pthread_mutex_lock(&mutex);
4.          while (storage < STORAGE_MAX)
5.              pthread_cond_wait(&cond, &mutex);
6.          storage = 0;
7.          pthread_mutex_unlock(&mutex);
8.      }
9.   }

10.  void *producer(void *args) {
11.     while (1) {
12.         sleep(1);
13.         storage++;
14.         pthread_mutex_lock(&mutex);
15.         if (storage >= STORAGE_MAX)
16.             pthread_cond_signal(&cond);
17.         pthread_mutex_unlock(&mutex);
18.     }
19.  }
```

Барьеры

Создание, удаление и ожидание

```
1.   int pthread_barrier_init(pthread_barrier_t *bp,
2.                            pthread_barrierattr_t *attr,
3.                            unsigned count);
4.   int pthread_barrier_destroy(pthread_barrier_t *bp);


5.   int pthread_barrier_wait(pthread_barrier_t *bp);
```

Барьеры

Пример

```
1.   int a = 0;
2.   pthread_barrier_t bar;
3.   void *f(void *b) {
4.      a++;
5.      pthread_barrier_wait(&bar);
6.      printf("%d", a); // ?
7.   }

8.   void main() {
9.      pthread_barrier_init(&bar, NULL, 3);
10.     pthread_create(&t1, NULL, f, NULL);
11.     pthread_create(&t2, NULL, f, NULL);
12.     pthread_create(&t3, NULL, f, NULL);
13.     // ...
14.  }
```

C++11: Мьютексы

Обычный мьютекс

```
1.  std::mutex Mutex;

2.  Mutex.lock();
3.  // ...
4.  Mutex.unlock();
```

```
1.   if (Mutex.try_lock()) {
2.      // ...
3.      Mutex.unlock();
4.   }
```

C++11: Мьютексы

Использование unique_lock

```
1.  std::mutex Mutex;

2.  void Something(void)
3.  {
4.      std::unique_lock<std::mutex> Locker(Mutex);
5.      // ...
6.  }
```

C++11: Мьютексы

Найди ошибку?

```
1.   class Container {
2.      std::mutex Mutex;
3.      std::list<int> L;
4.   public:
5.      void add(int el) {
6.          Mutex.lock();
7.          L.push_back(el);
8.          Mutex.unlock();
9.      }
10.     void add_many(int n, ...) {
11.         va_list arguments;
12.         va_start(arguments, num);
13.         Mutex.lock();
14.         for (int i = 0; i < n; ++i)
15.             add(va_arg(arg, int));
16.         Mutex.unlock();
17.         va_end(arguments);
18.     }
19.  };
```

C++11: Мьютексы

Рекурсивный мьютекс

```
1.   class Container {
2.      std::recursive_mutex Mutex;
3.      std::list<int> L;
4.   public:
5.      void add(int el) {
6.          Mutex.lock();
7.          L.push_back(el);
8.          Mutex.unlock();
9.      }
10.     void add_many(int n, ...) {
11.         va_list arguments;
12.         va_start(arguments, num);
13.         Mutex.lock();
14.         for (int i = 0; i < n; ++i)
15.             add(va_arg(arg, int));
16.         Mutex.unlock();
17.         va_end(arguments);
18.     }
19.  };
```

C++11: Мьютексы

Timed-мьютексы

```
1.  std::timed_mutex Mutex1;
2.  std::recursive_timed_mutex Mutex2;
```

```
3.  try_lock_for
4.  try_lock_until
```

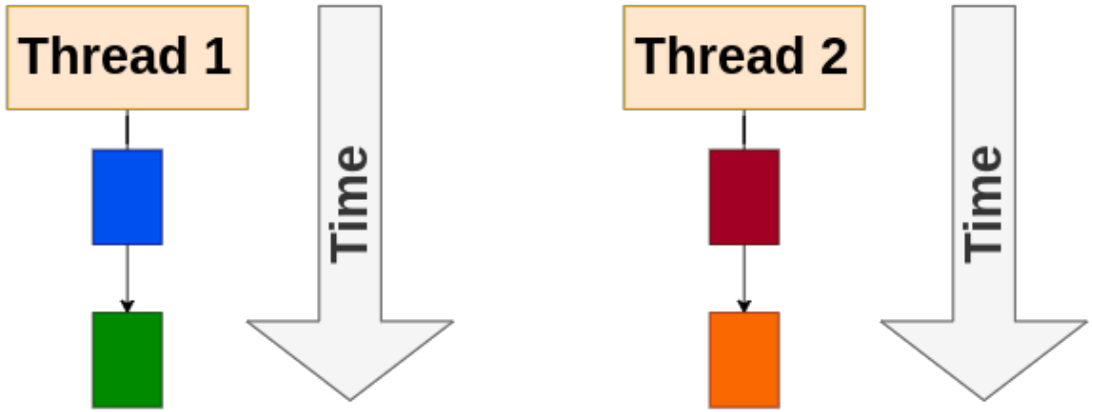
C++11: Условные переменные

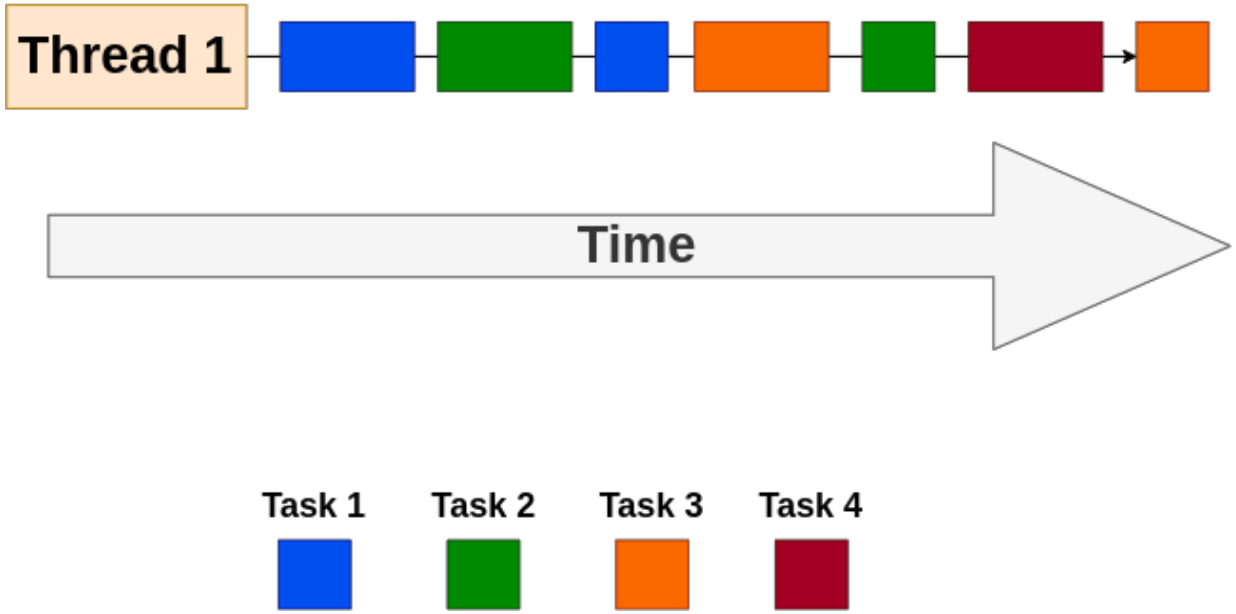
Пример на условные переменные

```
1.  std::mutex Mutex;
2.  std::condition_variable Cond;

3.  void print_id(int id)
4.  {
5.      std::unique_lock<std::mutex> Lock(Mutex);
6.      Cond.wait(Lock);
7.      // ...
8.  }

9.  void go()
10. {
11.     std::unique_lock<std::mutex> Lock(Mutex);
12.     Cond.notify_all();
13. }
```

Асинхронность

Синхронность
Асинхронность





C++11: Асинхронные функции

Пример на std::future и std::async

```
1.   bool my_function(int x) {
2.      // ...
3.      return true;
4.   }

5.   int main() {
6.      std::future<bool> future = std::async(std::launch::async | std::launch::deffered,
7.                                            my_function,
8.                                            16);
9.      // ...
10.     bool result = future.get()
11.     // ...
12.  }
```

C++11: Асинхронные функции

Пример использования std::promise и std::future

```
1.  result_type my_function() {
2.      return result_of_some_complex_comtutations();
3.  }

4.  std::future<result_type> async_deferred(result_type (*func)()) {
5.     std::promise<result_type> promise;
6.     std::future<result_type> future = promise.get_future();

7.     std::thread thread([](std::promise<result_type> &&promise, result_type (*func)()) {
8.         try {
9.             promise.set_value(func());
10.        } catch (...) {
11.            promise.set_exception(std::current_exception());
12.        }
13.    }, std::move(promise), func);

14.    thread.detach();
15.    return future;
16. }

17. auto future = async_deferred(my_function);
18. result_type result_of_my_function = future.get();
```