Краткий курс лекций

ОПЕРАЦИОННЫЕ СИСТЕМЫ ДЛЯ РАЗРАБОТЧИКОВ ПО

ЛЕКЦИЯ №2

ДВФУ

к.т.н. Боровик Алексей Игоревич

План курса

- Введение
  - Что такое ОС? Зачем они нужны?
  - Основные идеи и принципы ОС
  - Ядро ОС, планировщик, прерывания, многозадачность
- Процессы, потоки и таймеры
  - Многозадачность
  - Процессы, потоки, средства IPC в Windows и POSIX
  - Работа с таймерами и временем в Windows и POSIX
  - Средства разработки кроссплатформенных приложений
- Сеть
  - Принцип построения сетей, стек протоколов TCP\IP
  - Интерфейсы создания сетевых приложений Windows и POSIX
  - Маршаллинг данных, средства RPC

План лекции

- Многозадачность
  - Понятие и виды многозадачности
  - Потоки и процессы
  - IPC в Windows и POSIX
    - Средства IPC
    - Механизмы синхронизации
- Таймеры и время
  - Особенности таймеров ОС
  - Работа со временем и календарем в Windows и POSIX
- Разработка кроссплатформенных приложений на C/C++
  - Предопределенные макросы компиляторов
  - Средства автоматизации сборки
  - Функции библиотек Boost и QT для реализации IPC и работы со временем

Перед началом…

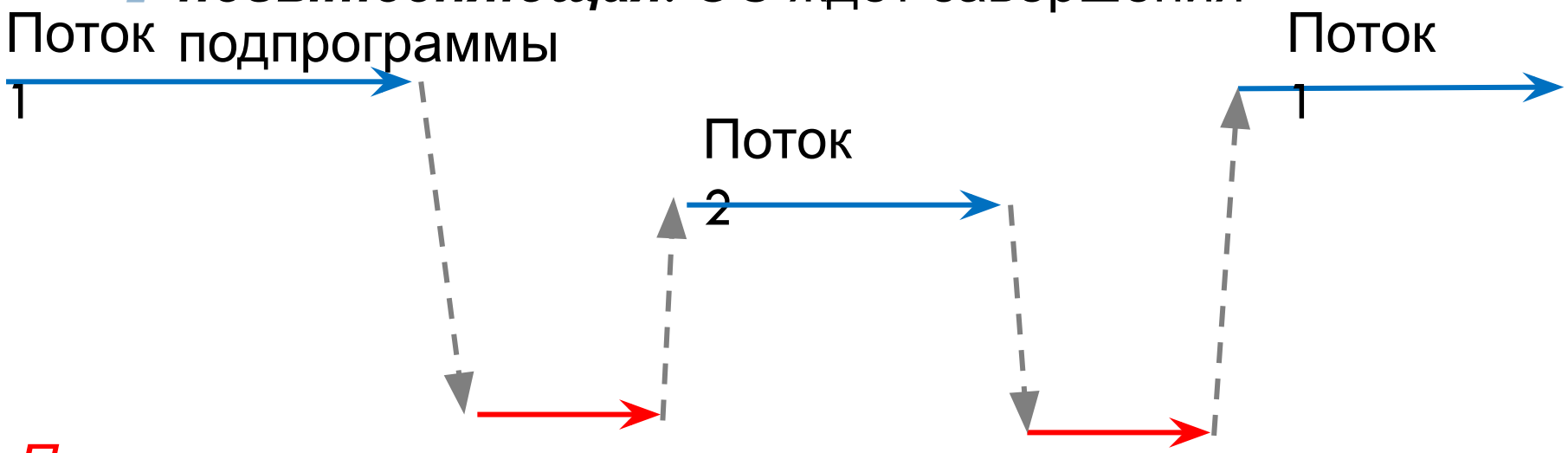
- Где смотреть описание функций API ОС?
  - для Microsoft Windows:
    - MSDN (Microsoft Developer Network):
    - https://docs.microsoft.com/en-us/windows/win32/
  - для POSIX:
    - The Open Group Base Specifications Issue:
    - https://pubs.opengroup.org/onlinepubs/9699919799/
  - для Linux:
    - Linux man pages online:
    - https://man7.org/linux/man-pages/index.html
  - **BONUS!**
    - В нашей директории на ЯДиске

Многозадачность

Многозадачность, типы, потоки и процессы, механизмы синхронизации: мьютекс, семафор, барьер, IPC

Многозадачность

- Одновременное выполнение нескольких подпрограмм (потоков)
- ОС сама переключает подпрограммы
  - ***вытесняющая***: ОС не ждёт завершения подпрограммы
  - ***невытесняющая***: ОС ждёт завершения подпрограммы

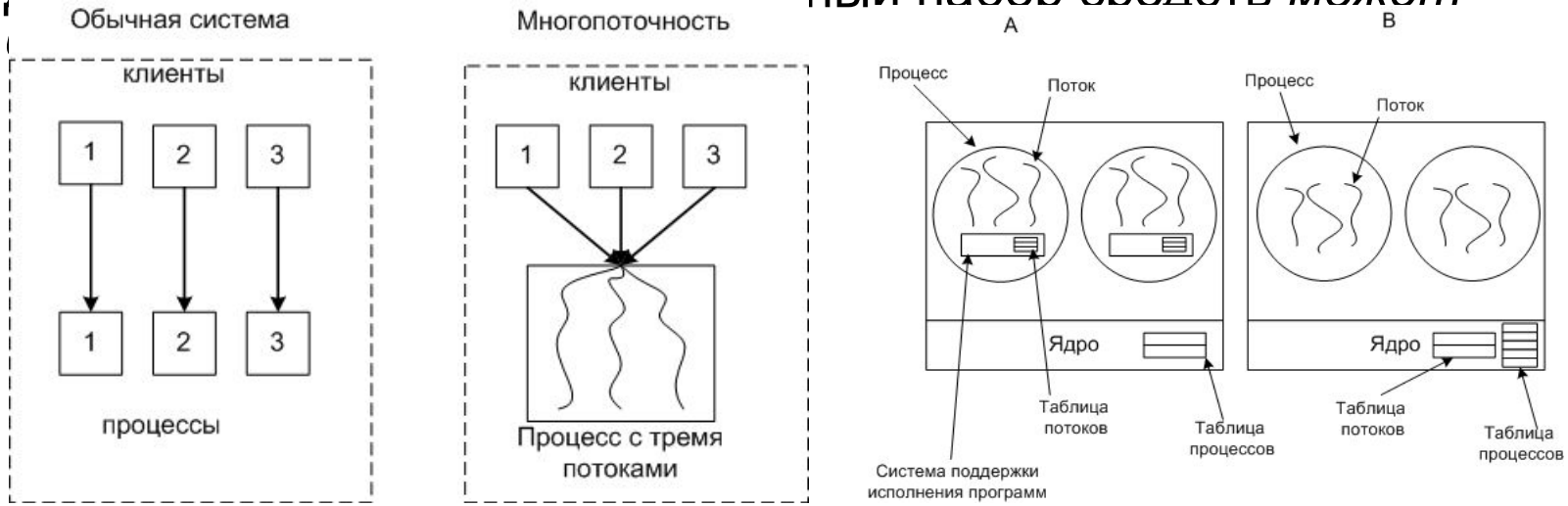


Многозадачность

- Невытесняющая многозадачность (tickless-система)
  - *совместная*, *кооперативная* многозадачность
  - планировщик вызывается по окончанию очередной задачи
  - (-) одна «повисшая» задача блокирует остальные
  - (+) пониженный расход энергии
  - (+) легко программировать
  - применяется в большинстве современных ОС МК
- Вытесняющая многозадачность
  - планировщик вызывается по прерыванию таймера
  - (+) одна «повисшая» задача не останавливает остальные
    - надежность системы значительно выше
  - (-) повышенный расход энергии
  - (-) необходимость использовать *механизмы синхронизации*, *принципы реентрабельности* и *потоковой безопасности*
  - применяется в большинстве ОС современных компьютеров и мобильных устройств

Потоки и процессы

- **Процесс** – выполняется ***в отдельном виртуальном адресном пространстве*** и имеет приоритет исполнения.
- **Поток (нить) исполнения** – выполняется ***в общем адресном пространстве процесса*** и, в некоторых ОС, *может иметь* приоритет исполнения.
- В большинстве ОС понятия поток (нить) и процесс *неравнозначны*.
- Для контроля доступа к *общей памяти* необходимо использовать средства IPC и\или механизмы синхронизации. Для потоков и процессов доступный набор средств *может*



Создание процесса POSIX

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
  pid_t pid;
  int rv;
  switch(pid=fork()) {
  case -1:
        perror("fork"); /* произошла ошибка */
        exit(1); /*выход из родительского процесса*/
  case 0:
        printf(" CHILD: Это процесс-потомок!\n");
        printf(" CHILD: Мой PID -- %d\n", getpid());
        printf(" CHILD: PID моего родителя -- %d\n",
            getppid());
        printf(" CHILD: Введите мой код возврата
                       (как можно меньше):");
        scanf(" %d");
        printf(" CHILD: Выход!\n");
        exit(rv);
  default:
        printf("PARENT: Это процесс-родитель!\n");
        printf("PARENT: Мой PID -- %d\n", getpid());
        printf("PARENT: PID моего потомка %d\n",pid);
        printf("PARENT: Я жду, пока потомок
                        не вызовет exit()...\n");
        wait();
        printf("PARENT: Код возврата потомка:%d\n",
                 WEXITSTATUS(rv));
        printf("PARENT: Выход!\n");
  }
}
```

Замещение тела процесса POSIX

```
1  #include <unistd.h>
2  int execl(char *name, char *arg0, ... /*NULL*/);
3  int execv(char *name, char *argv[]);
4  int execle(char *name, char *arg0, ... /*,NULL, char *envp[]*/);
5  int execve(char *name, char *arv[], char *envp[]);
6  int execlp(char *name, char *arg0, ... /*NULL*/);
7  int execvp(char *name, char *argv[]);
```

- Новая программа загружается в память вместо старой, вызвавшей exec(). Старой программе больше недоступны сегменты памяти – они перезаписаны новой программой!
- Функции с именем, оканчивающимся на **e** позволяют задать новый список переменных окружения, вместо стандартных
- Доступ к переменным окружения:
  - **int** main(**int** argc, **char** *argv[], **char** * envp[]);
  - **<stdlib.h>**: **char** * getenv(const char *name);
  - ***<unistd.h>***: **extern char** ** *environ*;

POSIX: функция system()

- **<stdlib.h>**: **int** system(**const char** *command);

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int system(char const *cmd)
{
    int pid, status;
    if ((pid = fork()) < 0) {
        /* ошибка */
        perror("fork");
        return -1;
    } else if (!pid) {
        /* child */
        execl("/bin/sh", "/bin/sh", "-c", cmd, NULL);
        /* ошибка */
        perror("execl");
        _exit(1);
    }
    /* parent */
    wait(&status);
    if (WIFSIGNALED(status))
        return WTERMSIG(status) + 256;
    return WEXITSTATUS(status);
}
```

POSIX: функция spawn()

- Функция spawn() запускает исполнимый файл и передает управление обратно вызвавшему процессу

```
#include <spawn.h>

int posix_spawn(pid_t *restrict pid, const char *restrict path,
                const posix_spawn_file_actions_t *restrict file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict],
                char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
                const posix_spawn_file_actions_t *restrict file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict],
                char *const envp[restrict]);
```

- Подождать завершение созданного процесса можно через **waitpid()**

Создание процесса Windows

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

void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 )
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Создание процесса Windows

- Можно, как и в POSIX, использовать функцию **system()** (объявлена в **<process.h>**):

```
int system(
   const char *command
);
int _wsystem(
   const wchar_t *command
);
```

- Нужно учитывать, что исполняться содержимое будет в Windows console:
  - EXE-файлы
  - BAT и CMD скрипты
  - Список команд

Создание потока в POSIX

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

void* helloWorld(void *args) {
    int* ptr = (int*)args;
    printf("Hello from thread: %d!\n",*ptr);
    return 0; // same as pthread_exit()
}

int main() {
    pthread_t thread;
    int status;
    int status_addr;

    int param = 1;
    status = pthread_create(&thread, NULL, helloWorld, &param);
    if (status != 0) {
        printf("main error: can't create thread, status = %d\n", status);
        exit(-1);
    }
    printf("Hello from main!\n");

    status = pthread_join(thread, (void**)&status_addr);
    if (status != SUCCESS) {
        printf("main error: can't join thread, status = %d\n", status);
        exit(-2);
    }

    printf("joined with address %d\n", status_addr);
    _getch();
    return 0;
}
```

Управление потоками POSIX

```
1  // В потоке-родителе можно использовать функции:
2  // Прервать исполнение указанного потока
3  int pthread_cancel(pthread_t thread);
4  // Послать потоку сигнал (убить поток)
5  #include <signal.h>
6  int pthread_kill(pthread_t thread, int sig);
7
8  // В потоке-наследнике можно использовать функции:
9  #include <pthread.h>
10 // Управлять возможностью прервать поток
11 int pthread_setcancelstate(int state, int *oldstate);
12 // Возможные значения state:
13 // PTHREAD_CANCEL_ENABLE и PTHREAD_CANCEL_DISABLE
14 // Установить точку выхода из потока
15 void pthread_testcancel(void);
16 // Задать функцию, которая выполнится при выходе из потока
17 void pthread_cleanup_push(void (*routine)(void*), void *arg);
18 // Убрать верхнюю функцию стека выхода и выполнить ее если нужно
19 void pthread_cleanup_pop(int execute);
```

Создание потока Windows

```c
#include <windows.h>
#define MAX_THREADS 3
#define BUF_SIZE 255
// Sample custom data structure for threads to use.
typedef struct MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

DWORD WINAPI MyThreadFunction(LPVOID lpParam)
{
    PMYDATA pDataArray = (PMYDATA)lpParam;
    printf("Hello from thread: %d!\n",pDataArray->val1);
    return 0;
}

int _tmain()
{
    PMYDATA pDataArray[MAX_THREADS];
    DWORD   dwThreadIdArray[MAX_THREADS];
    HANDLE  hThreadArray[MAX_THREADS];

    // Create MAX_THREADS worker threads.
    for(int i = 0; i < MAX_THREADS; i++) {
        // Allocate memory for thread data
        pDataArray[i] = (PMYDATA) HeapAlloc(GetProcessHeap(),
                                            HEAP_ZERO_MEMORY, sizeof(MYDATA));
        // Generate unique data for each thread to work with
        pDataArray[i]->val1 = i; pDataArray[i]->val2 = i+100;
```

```c
        // Create the thread to begin execution on its own.
        hThreadArray[i] = CreateThread(
            NULL,                   // default security attributes
            0,                      // use default stack size
            MyThreadFunction,       // thread function name
            pDataArray[i],          // argument to thread function
            0,                      // use default creation flags
            &dwThreadIdArray[i]);   // returns the thread identifier
        // Check the return value for success.
        // If CreateThread fails, terminate execution.
        // This will automatically clean up threads and memory.
        if (hThreadArray[i] == NULL) {
           printf("Error creating thread!");
           ExitProcess(3);
        }
    } // End of main thread creation loop.
    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, hThreadArray, TRUE, INFINITE);

    // Close all thread handles and free memory allocations.
    for(int i=0; i < MAX_THREADS; i++) {
        CloseHandle(hThreadArray[i]);
        if(pDataArray[i] != NULL) {
            HeapFree(GetProcessHeap(), 0, pDataArray[i]);
            pDataArray[i] = NULL;    // Ensure address is not reused.
        }
    }
    return 0;
}
```

Управление потоками Windows

```
// Со стороны родительского потока:
// 1.1 Создать событие и поделиться HANDLE события с потоком
HANDLE CreateEvent(
    LPSECURITY_ATTRIBUTES lpEventAttributes, // pointer to security attributes
    BOOL bManualReset,                       // flag for manual-reset event
    BOOL bInitialState,                      // flag for initial state
    LPCTSTR lpName                           // pointer to event-object name
);
// Пример:
HANDLE _cleanup_event = CreateEvent(NULL,FALSE,FALSE,NULL);
// 1.2 Установить событие в состояние "Сигнализировано"
SetEvent(_cleanup_event);
// 2.1 Терминировать поток
BOOL TerminateThread(
    HANDLE hThread, // handle to the thread
    DWORD dwExitCode // exit code for the thread
);
// Со стороны порожденного потока:
// 1.1 Организовать работу основной функции в блоке try..catch
try{
    thread->Main();
}
catch (TermEx) {}
// 2.1 В месте, где можно прервать работу потока - проверить событие
if (WaitForSingleObject(_cleanup_event,0) == WAIT_OBJECT_0) {
    throw TermEx(0);
}
```

IPC

- Межпроцессное взаимодействие (**inter-process communication, IPC**) — обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых *ядром ОС* или процессом, использующим *механизмы ОС* и реализующим новые возможности IPC. Может осуществляться как на одном компьютере, так и *между несколькими компьютерами сети*.
  - Файл (все ОС)
  - Сигнал (большинство ОС, но не Windows)
  - Неименованный канал (POSIX, Windows)
  - Именованный канал (POSIX, Windows)
  - Разделяемая память (POSIX, Windows)
  - ***Сокет*** (большинство ОС)
  - ***Обмен сообщениями*** (сторонние относительно ОС средства)

Работа с файлами

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    FILE *S1, *S2;
    int x, y;
    printf("Введите число : ");
    scanf("%d", &x);
    S1 = fopen("S1.txt", "w");
    if (S1 == NULL) {
        printf("Нет такого файла!");
        return -1;
    }
    fprintf(S1, "%d", x);
    fclose(S1);
    S1 = fopen("S1.txt", "r");
    S2 = fopen("S2.txt", "w");
    fscanf(S1, "%d", &y);
    y += 3;
    fclose(S1);
    fprintf(S2, "%d\n", y);
    fclose(S2);
    return 0;
}
```

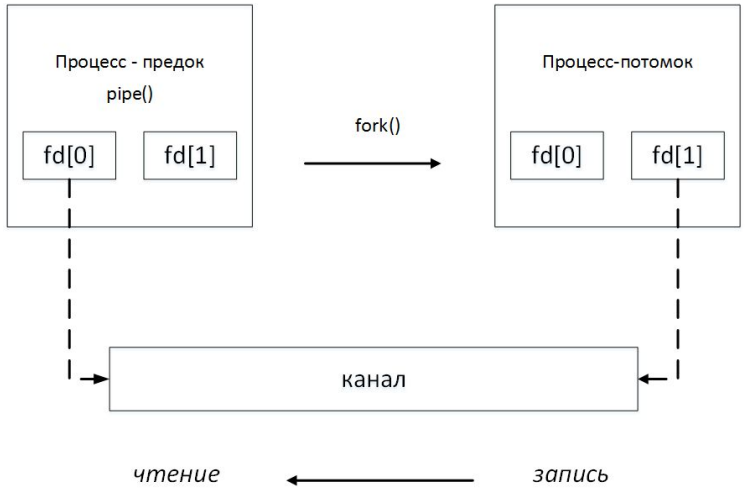
Сигналы в POSIX

- Сигналы могут быть посланы:
  - ядром системы для информирования приложения об ошибках или событиях ввода-вывода
  - пользователем из терминала, по нажатию специальных комбинаций клавиш: CTRL+C, CTRL+Z и т.п.
  - из другого приложения:
    - **int kill(pid_t *pid*, int *sig*);**
- Каждый процесс имеет свою маску сигналов (игнорируемые сигналы) и может задавать обработчики сигналов
  - Обработчик по-умолчанию закрывает программу
  - Потокам можно отправлять сигналы из основного процесса, у них есть маски, но нет возможности задать обработчик
  - Внутри обработчика сигнала безопасно можно менять только значения переменных, объявленных с ключевым словом volatile

```c
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
volatile unsigned char need_exit = 0;
void usr_sign_handler(int sig)
{
    puts("SIGNAL HANDLER!");
    if (sig == SIGUSR1)
        need_exit = 1;
}
int main(int argc, char** argv)
{
    struct sigaction act;
    memset(&act, 0, sizeof(act));
    act.sa_handler = usr_sign_handler;
    sigset_t   set;
    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    sigaddset(&set, SIGUSR2);
    act.sa_mask = set;
    sigaction(SIGUSR1, &act, NULL);
    sigaction(SIGUSR2, &act, NULL);
    for(;!need_exit;usleep(1e6))
        puts("I am still breathing!");
}
```

Неименованный канал (pipe)

- Доступен только связанным процессам – родительскому и дочернему
- Использует стратегию работы с данными FIFO
  - Прочитанная информация немедленно удаляется из канала
  - При чтении из пустого канала процесс блокируется до поступления данных
- Неименованный канал создается:
  - POSIX: **int pipe(int *fildes*[2])** из **<unistd.h>**
  - Windows:
    - BOOL **CreatePipe**(PHANDLE *hReadPipe*, PHANDLE *hWritePipe*, LPSECURITY_ATTRIBUTES *lpPipeAttributes*, DWORD *nSize*);
  - Для чтения и записи из\в канал можно использовать обычные функции работы с файлами:
    - POSIX: read(), write() из <unistd.h>
    - Windows: ReadFile() WriteFile()

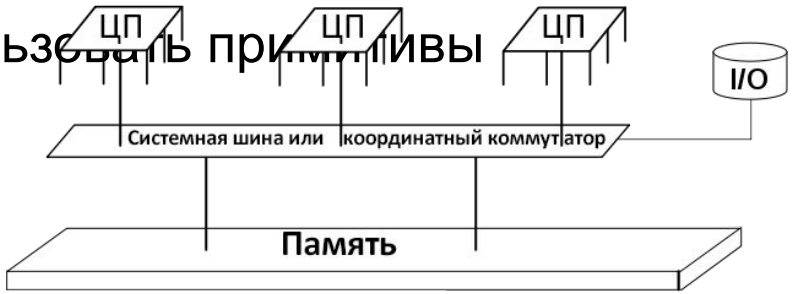


Именованный канал (named pipe)

- В POSIX именованный канал существует независимо от использующих его процессов и имеет имя в системе.
  - Для создания канала используется
    - **int mkfifo(const char *path, mode_t mode)** (из **<sys/stat.h>**)
    - программа **mkfifo**
    - работать с каналом можно как с обычным файлом (**fopen**(), **fread**(), **fwrite**())
  - Для удаления канала нужно использовать
    - **int remove(const char *path);**
    - программа **rm**
  - Именованные каналы можно использовать с перенаправлением ввода\вывода и любыми программами, которые работают с обычными файлами и\или потоками
- В Windows именованные каналы организуют клиент-серверное взаимодействие. Имеют имя \\.\pipe\имя и удаляются, когда никто их не использует
  - Создание канала: **CreateNamedPipe**()
  - Ожидание подключения на стороне сервера: **ConnectNamedPipe**()
  - На стороне клиента: **CreateFile**(), **CloseHandle**(), **ReadFile**(), **WriteFile**() или функции **fopen**(), **fclose**(), **fread**(), **fwrite**()
  - Пример сервера:
    - https://docs.microsoft.com/en-us/windows/win32/ipc/named-pipe-server-using-overlapped-i-o
  - Пример клиента:

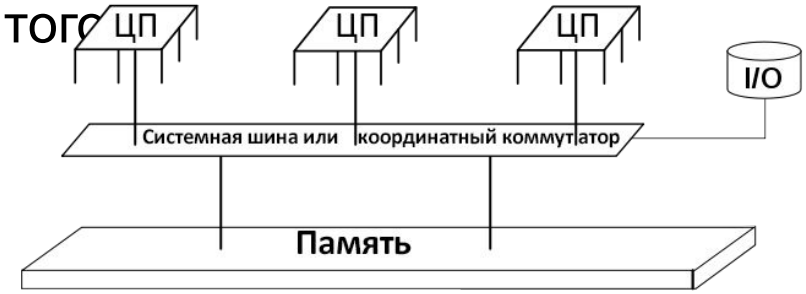
Разделяемая память в POSIX

- Область памяти, одновременно доступная в разных процессах
- Самый быстрый IPC
- В POSIX для работы используются:
  - **#include <sys/mman.h>**
  - **shm_open**() – связать область памяти с файловым дескриптором
  - **ftruncate()** из **<unistd.h>** для изменения размера области
  - **mmap()** для подключения разделяемой памяти к адресному пространству процесса
    - после подключения с памятью можно работать через обычный указатель
  - обязательно необходимо использовать примитивы синхронизации – семафоры
  - https://habr.com/ru/post/122108/



Разделяемая память в Windows

- **CreateFileMapping**() для создания глобально именованной области, с указанием размера
- **OpenFileMapping**() для открытия глобально именованной области памяти
- **MapViewOfFile**() для подключения памяти к адресному пространству процесса и получения указателя на нее
- **UnmapViewOfFile**() для отключения памяти от локального адресного пространства
  - Когда последний процесс вызовет эту функцию, содержимое памяти очистится и объект удалится
- Для разграничения доступа можно использовать любые **именованные** примитивы синхронизации.
- **LsaAllocateSharedMemory**() – не для того



Сокеты Беркли

- Сокеты впервые появились в ОС Berkeley UNIX 4.2 BSD (1983 г)
- Сокет в POSIX-системе это «файл» специального вида
  - Все, что записывается в файл, передается по сети
  - Все, что получено из сети, можно прочитать из файла
  - Передача данных по сети скрыта от программиста
- Сокеты - де-факто стандарт интерфейсов для транспортной подсистемы
- Сокеты (разной реализации)

Операции с сокетами

Операция	Назначение
Socket	Создать новый сокет
Bind	Связать сокет с IP-адресом и портом
Listen	Объявить о желании принимать соединения
Accept	Принять запрос на установку соединения
Connect	Установить соединение
Send	Отправить данные по сети
Receive	Получить данные из сети
Close	Закрыть соединение

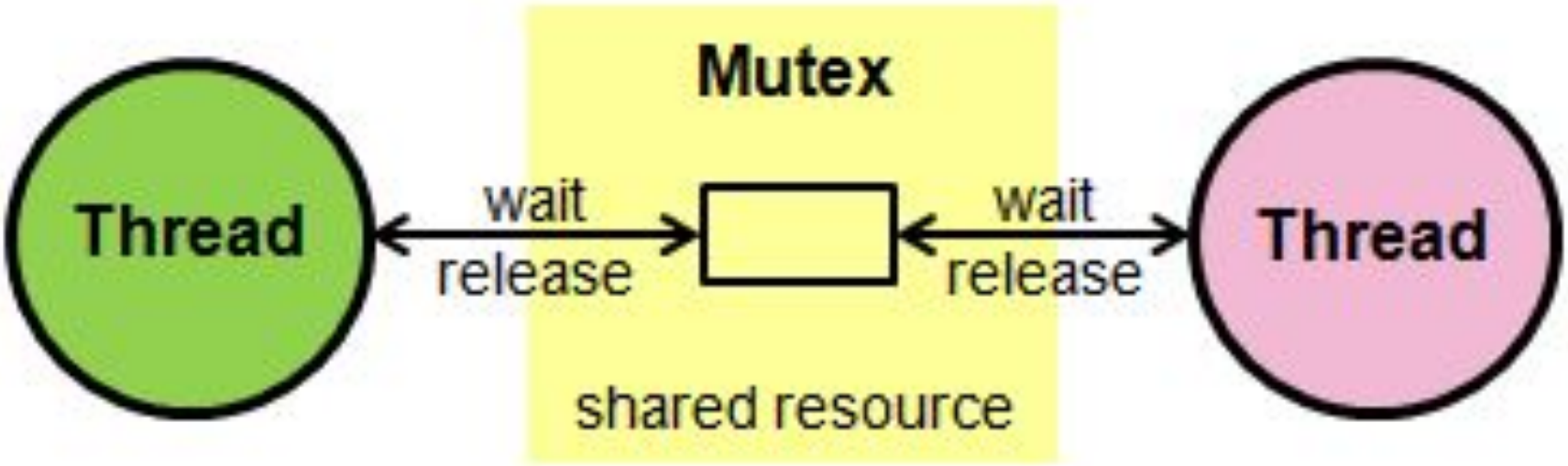
Сторонние средства IPC/RPC

- **DCE/RPC** (Distributed Computing Environment / Remote Procedure Calls) — бинарный протокол на базе различных транспортных протоколов, в том числе TCP/IP и Named Pipes из протокола SMB/CIFS
- **DCOM** (Distributed Component Object Model или «Network OLE») — объектно-ориентированное расширение DCE/RPC, позволяющее передавать ссылки на объекты и вызывать методы объектов через ссылки
- ZeroC **ICE**
- **JSON-RPC**— JavaScript Object Notation Remote Procedure Calls (текстовый протокол на базе HTTP) Спецификация RFC-4627
- **.NET Remoting** (бинарный протокол на базе TCP, UDP, HTTP)

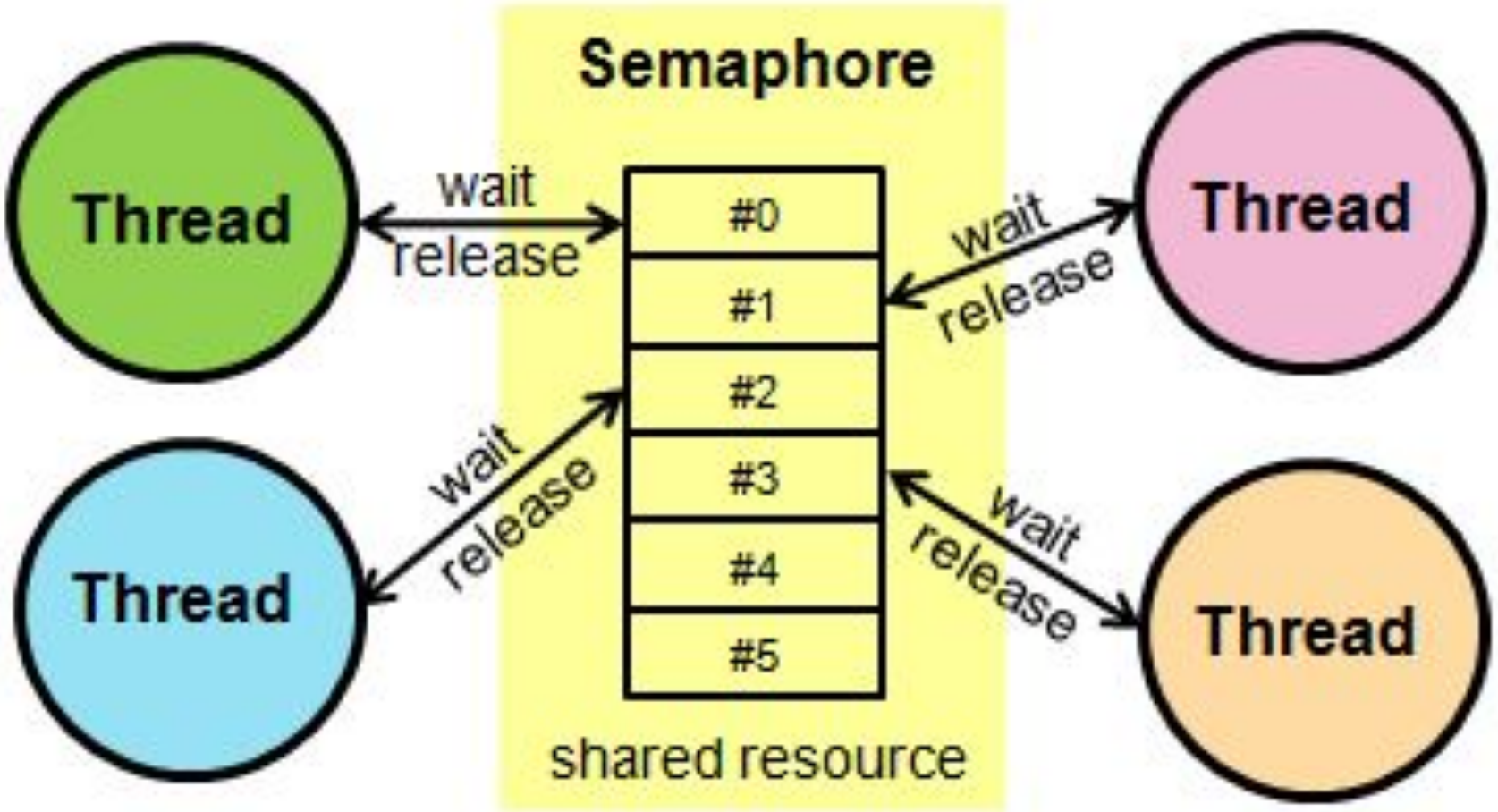
Механизмы синхронизации

- При работе с общей памятью из разных потоков или процессов необходимо использовать **механизмы синхронизации**.
- **Мьютекс** (Mutex = mutual exclusion) – обеспечивает доступ только одного потока к критическому участку кода; остальные потоки ждут освобождения мьютекса, а потом получают монопольный доступ к критическому участку, в соответствии со своим приоритетом.
- **Семафор** – атомарный счетчик, над которым можно выполнять две операции: уменьшение на 1 (захват) и увеличение на 1 (освобождение). При попытке уменьшения семафора, значение которого равно нулю, задача, запросившая данное действие, должна блокироваться до тех пор, пока не станет возможным уменьшение значения семафора до неотрицательного значения, то есть пока другой процесс не увеличит значение семафора.
- **Барьер** –механизм синхронизации критических точек у группы задач. Задачи могут пройти через барьер только все сразу. Перед входом в критические точки задачи группы должны блокироваться, пока входа в критическую точку не достигнет последняя задача из группы. Как только все задачи окажутся

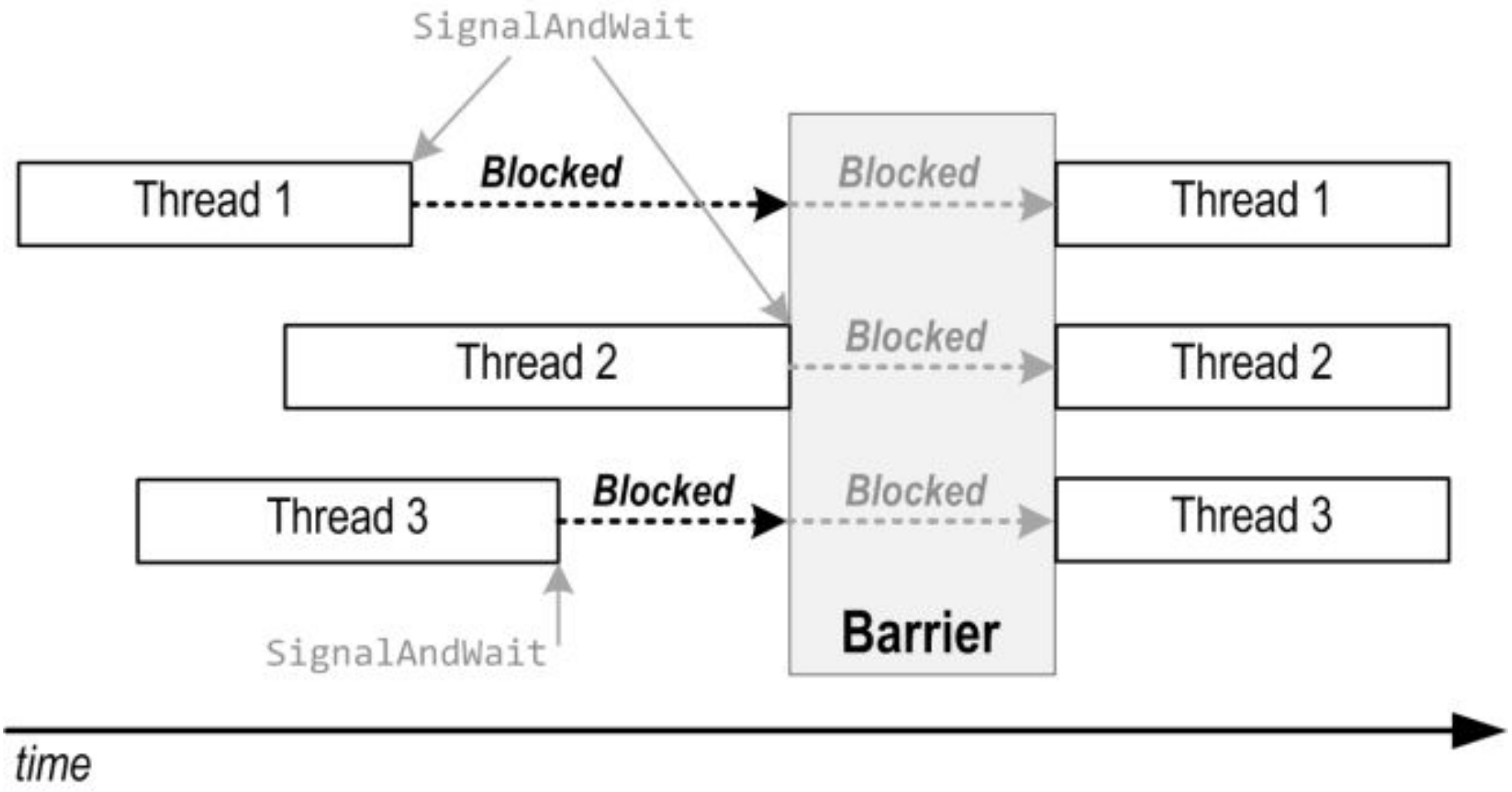
Mutex


Mutex
Thread
wait
release
wait
release
Thread
shared resource

Семафор


Semaphore
Thread
wait
release
#0
#1
#2
#3
#4
#5
Thread
wait
release
Thread
wait
release
Thread
wait
release
shared resource

Барьер


SignalAndWait
Thread 1
Blocked
Blocked
Thread 1
Thread 2
Blocked
Thread 2
Thread 3
Blocked
Blocked
Thread 3
SignalAndWait
Barrier
time

Синхронизация в POSIX

- Семафоры (**#include <semaphore.h>**) – для синхронизации процессов
  - Неименованный: **int sem_init(sem_t** *sem, **int** *pshared*, **unsigned** *value*);
  - Именованный: **sem_t *sem_open(const char** *name, **int** oflag, **mode_t** mode, **unsigned int** value);
    - */somename*
  - Работа с семафорами:
    - int sem_wait(sem_t **sem*); - заблокировать семафор
    - int sem_trywait(sem_t **sem*); - проверить семафор
    - int sem_post(sem_t **sem*); - разблокировать семафор
    - int sem_close(sem_t **sem*); - закрыть именованный семафор
    - int sem_destroy(sem_t **sem*); - закрыть неименованный семафор
- Мьютексы (**#include <pthread.h>**) – для синхронизации потоков
  - Создание и удаление:
    - int pthread_mutex_init(pthread_mutex_t *restrict *mutex*, const pthread_mutexattr_t *restrict *attr*);
    - int pthread_mutex_destroy(pthread_mutex_t **mutex*);
  - Работа:
    - int pthread_mutex_lock(pthread_mutex_t **mutex*);
      int pthread_mutex_trylock(pthread_mutex_t **mutex*);
      int pthread_mutex_unlock(pthread_mutex_t **mutex*);
- Условные переменные (condition variable) (**#include <pthread.h>**) - * потоков
  - Создание\удаление: pthread_cond_init() , pthread_cond_destroy()
  - Ожидание: pthread_cond_wait(), pthread_cond_timedwait()
  - Сигнализирование: pthread_cond_signal() , pthread_cond_broadcast()

Синхронизация в Windows

- Для синхронизации процессов используются именованные примитивы:
  - Мьютексы:
    - **HANDLE** *CreateMutex*(**LPSECURITY_ATTRIBUTES** lpMutexAttributes, **BOOL** bInitialOwner, **LPCTSTR** lpName);
    - **HANDLE** *OpenMutex*(**DWORD** dwDesiredAccess, **BOOL** bInheritHandle, **LPCTSTR** lpName);
    - **BOOL** *ReleaseMutex*(**HANDLE** hMutex);
  - Семафоры:
    - **HANDLE** *CreateSemaphore*(**LPSECURITY_ATTRIBUTES** lpSemaphoreAttributes, **LONG** lInitialCount, **LONG** lMaximumCount, **LPCTSTR** lpName);
    - **HANDLE** *OpenSemaphore*(**DWORD** dwDesiredAccess, **BOOL** bInheritHandle, **LPCTSTR** lpName);
    - **BOOL** *ReleaseSemaphore*(**HANDLE** hSemaphore, **LONG** lReleaseCount, **LPLONG** lpPreviousCount);
- Для синхронизации потоков можно использовать неименованные примитивы:
  - **VOID** *InitializeCriticalSection*(**LPCRITICAL_SECTION** lpCriticalSection);
  - **VOID** *DeleteCriticalSection*(**LPCRITICAL_SECTION** lpCriticalSection);
  - *EnterCriticalSection*(), *TryEnterCriticalSection*(), *LeaveCriticalSection*()
- Для ожидания мьютекса или семафора, а также реализации условной переменной нужно использовать:
  - **DWORD** *WaitForSingleObject*(**HANDLE** hHandle, **DWORD** dwMilliseconds);
  - **DWORD** *WaitForMultipleObjects*(**DWORD** nCount,**CONST HANDLE** *lpHandles, **BOOL** bWaitAll, **DWORD** dwMilliseconds);

Таймеры

Принцип устройства таймера, работа с датой и временем

Таймеры ОС

- Аппаратные таймеры
  - ограниченное число таймеров
  - всего два программируемых события (будильника) на один таймер
  - ограниченная глубина счёта таймера
- Таймер в ОС — это программный модуль
  - использует всего 1 аппаратный таймер (обычно, самый большой из доступных – 32 бита)
  - ведёт список всех запланированных задач
  - ставит будильник на ближайшую задачу
  - по срабатыванию – рассчитывает время до следующей задачи
  - переставляет будильник на следующую задачу
  - фиксирует моменты переполнения таймера и корректно их обрабатывает

Работа со временем в ОС

- Аппаратно время отсчитывается RTC (realtime clock)
  - В настольных компьютерах размещены на материнской плате
    - Микросхема счета
    - Кварцевый резонатор
    - Батарейка
- Любая ОС предоставляет функции для работы с датой
  - Обычно дата представлена в UNIX-time
    - количество секунд, прошедших с полуночи (00:00:00 UTC) 1 января 1970 года («эпоха Unix»)
- Любая ОС предоставляет функции для замораживания (ожидания таймера или события) потоков и процессов.
  - Исполнение замороженного процесса откладывается планировщиком до таймаута

Время в POSIX

- Ожидание (**#include <unistd.h>**):
  - unsigned *sleep*(unsigned *seconds*);
  - int *usleep*(useconds_t *useconds*);
  - **#include <time.h>**:
    - int *nanosleep*(const struct timespec *req, struct timespec *rem);
- Получить время:
  - **#include <time.h>**: time_t *time*(time_t *tloc);
  - **#include <sys/time.h>**:
    - int *gettimeofday*(struct timeval *restrict *tp*, void *restrict *tzp*);
- Работа с датой (**#include <time.h>**):
  - struct tm *localtime(const time_t *timer);
  - struct tm *gmtime(const time_t *timer);
  - size_t *strftime*(char *restrict *s*, size_t *maxsize*,
    const char *restrict *format*, const struct tm *restrict *timeptr*);
- Огромное количество других функций, например clock_* для работы с конкретными часами

Время в Windows

- Ожидание:
  - VOID Sleep(DWORD dwMilliseconds);
  - DWORD SleepEx(DWORD dwMilliseconds, BOOL bAlertable);
- Получить время:
  - **#include <time.h>**: time_t *time*(time_t *tloc);
  - GetSystemTimeAsFileTime:

```
union
{
    long long ns100; /* time since 1 Jan 1601 in 100ns units */
    FILETIME ft;
} now;
GetSystemTimeAsFileTime( &(now.ft) );
ts.tv_usec=(long)((now.ns100 / 10LL) % 1000000LL );
ts.tv_sec= (long)((now.ns100-(116444736000000000LL))/10000000LL);
```

- Получить дату:
  - VOID GetSystemTime(LPSYSTEMTIME lpSystemTime);
  - VOID GetLocalTime(LPSYSTEMTIME lpSystemTime);

Кроссплатформенность в C/C++

Предопределенные макросы компиляторов,
средства автоматизации сборки,
функции библиотек Boost и QT для реализации IPC и работы со временем

Макросы компиляторов

- Кроссплатформенный код на C/C++ обычно пишется с использованием макросов, определяющих ОС, компилятор, аппаратное обеспечение и т.п.
- Список предопределенных макросов: https://sourceforge.net/projects/predef/

```
// This one is for components in component libraries
// You can avoid using it if you won't compile your components under Visual Studio
// But it is strictly recommended for you to include this macros
#if defined (_MSC_VER) && defined (RCE_COMPONENT_LIBRARY_CODE)
#   if defined (RCEComponentLibrary_STATIC)
#       define RCE_COMPONENT_EXPORT
#   elif defined (RCEComponentLibrary_EXPORTS)
#       define RCE_COMPONENT_EXPORT __declspec (dllexport)
#   else
#       define RCE_COMPONENT_EXPORT __declspec (dllimport)
#   endif
#else
#   define RCE_COMPONENT_EXPORT
#endif

// GNU compiler version
#if defined(__GNUC__)
#   if defined(__GNUC_PATCHLEVEL__)
#       define __GNUC_VERSION__ (__GNUC__ * 10000 \
                               + __GNUC_MINOR__ * 100 \
                               + __GNUC_PATCHLEVEL__)
#   else
#       define __GNUC_VERSION__ (__GNUC__ * 10000 \
                               + __GNUC_MINOR__ * 100)
#   endif
```

Example

VxWorks	_WRS_VXWORKS_MAJOR	_WRS_VXWORKS_MINOR	_WRS_VXWORKS_MAINT
6.2	6	2	0

Windows

Type	Macro	Description
Identification	_WIN16	Defined for 16-bit environments 1
Identification	_WIN32	Defined for both 32-bit and 64-bit environments 1
Identification	_WIN64	Defined for 64-bit environments 1
Identification	__WIN32__	Defined by Borland C++
Identification	__TOS_WIN__	Defined by xlC
Identification	__WINDOWS__	Defined by Watcom C/C++

Windows CE

Type	Macro	Format	Description
Identification	_WIN32_WCE		Defined by Embedded Visual Studio C++

Автоматизация сборки

- Система автоматизации сборки решает множество задач разработки ПО:
  - Компиляция объектных модулей
    - Определение ОС или доступности тех или иных модулей
  - Линковка объектных модулей в исполняемые файлы
    - Поиск зависимостей
  - Выполнение тестов
  - Развертывание системы в целевой среде
  - Автоматическое создание документации программиста, описание изменений
- Популярные системы автоматизации сборки:
  - Make (только POSIX системы)
  - SCons (https://scons.org/)
  - **CMake** (https://cmake.org/)
  - **QMake** (поставляется с QT)

Boost и QT

- Boost.Threads
- Boost.Process
- Boost.Interprocess
- Boost.Filesystem
- Boost.Date_Time

- QThread
- QProcess
- QSharedMemory, QTcpSocket, QTcpServer,…
- QFile
- QDateTime

- https://www.boost.org/doc/libs/1_78_0/?view=categorized
- https://doc.qt.io/qt-5/index.html

C++11 и C++17

- C++11:
  - std::thread из <thread>
  - std::mutex, std::recursive_mutex, std::condition_variable
  - std::shared_ptr
  - std::atomic<>
- C++17:
  - std::filesystem из boost::filesystem
- Далее:
  - Возможно, появятся и сокеты ☺

Следующая лекция

Работа с сетью в ОС

- *Принципы построения сетей, стек протоколов TCP\IP*
- *Интерфейсы создания сетевых приложений в Windows и POSIX*
- *Особенность обмена бинарными данными, маршаллинг данных*
- *Средства RPC*

Не переключайтесь…