Многопоточность

Таранцев Игорь Геннадьевич
Лаб.13 ИАиЭ СО РАН
СофтЛаб-НСК

Многозадачность

- Однозадачность
- Корпоративная многозадачность
- Вытесняющая многозадачность
- Многопоточность

Многопоточность

Эффективное использование ресурсов

Видеоплейер:

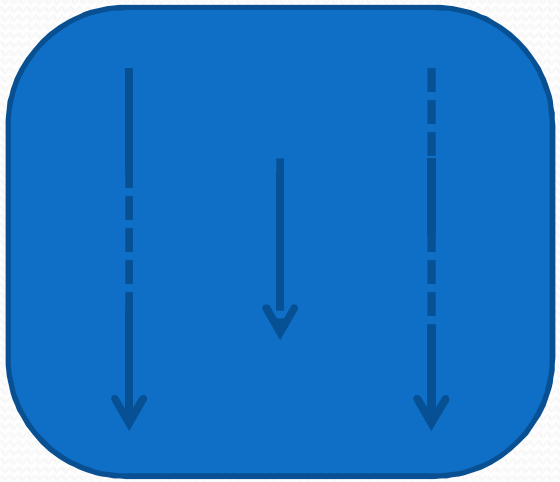
- Чтение файла (файловая система)
- Декодирование (CPU)
- Показ (графика)

Многопоточность

- Одновременность выполнения потоков:
  - Временная многопоточность (англ. Temporal multithreading)
  - Одновременная многопоточность (англ. Simultaneous multithreading)
- Приоритет потока:
  - Фиксированный приоритет
  - Приоритет как стартовое значение счетчика

Многопоточность

- Процесс
  - Адресное пространство
  - Память
  - Файловые дескрипторы
- Поток
  - SP / PC / регистры
  - Стек
  - Специальные данные (run-time)



Средства синхронизации

- Взаимоисключения (Mutex)
- Критические секции
- События
- Семафоры
- Условные переменные
- Порты завершения ввода-вывода (IOCP IO completion port)
  PIPE, file, IP-port…
- Shared Mutex

Примитивы C++ (v.11)

- thread – поток управления
- mutex – защита данных от одновременного доступа или защита кода от одновременного исполнения
- condition_variable – условная переменная (передача сигналов между потоками)
- future/promise/packaged_task/… – передача/ожидание состояния завершения асинхронных задач
- atomic… – транзакционность чтения/модификации/записи данных

std::thread

- Конструктор:
  - std::thread t1(f1);
  - std::thread t2(f2, 10);
  - std::thread t3(f3, std::ref(v3));
- Деструктор убивает поток !!!
- tx.join() - ждать завершения потока
- tx.detach() – «отпустить поток в свободное плавание»

std::mutex

- Пустой конструктор и деструктор

- **`mutex.lock()`** – захватить объект (ждать, если захвачен кем-то другим).
- **`mutex.unlock()`** – освободить объект.

- **`bool mutex.try_lock()`** – захватить объект, если он свободен и вернуть true, иначе вернуть false.

std::mutex

- защита данных от одновременного доступа
- защита кода от одновременного исполнения

```
std::mutex g_mMyData;
class MyData g_MyData;
```

```
g_mMyData.lock();
g_MyData = ...;
g_mMyData.unlock();
```

Поток №1

```
g_mMyData.lock();
... = g_MyData;
g_mMyData.unlock();
```

Поток №2

std::mutex

- std::recursive_mutex – разрешает повторный захват из того же потока
- std::timed_mutex – умеет ждать освобождения объекта в течении заданного времени:

```
bool try_lock_for(duration);
```

- std::recursive_timed_mutex – комбинация первых двух вариантов

std::mutex (C++ v.17)

- std::shared_mutex – два типа захвата:
  - shared (захватывают много потоков сразу)
    `lock_shared / try_lock_shared / unlock_shared`
  - exclusive (захватывает только один поток)
    `lock / try_lock / unlock`

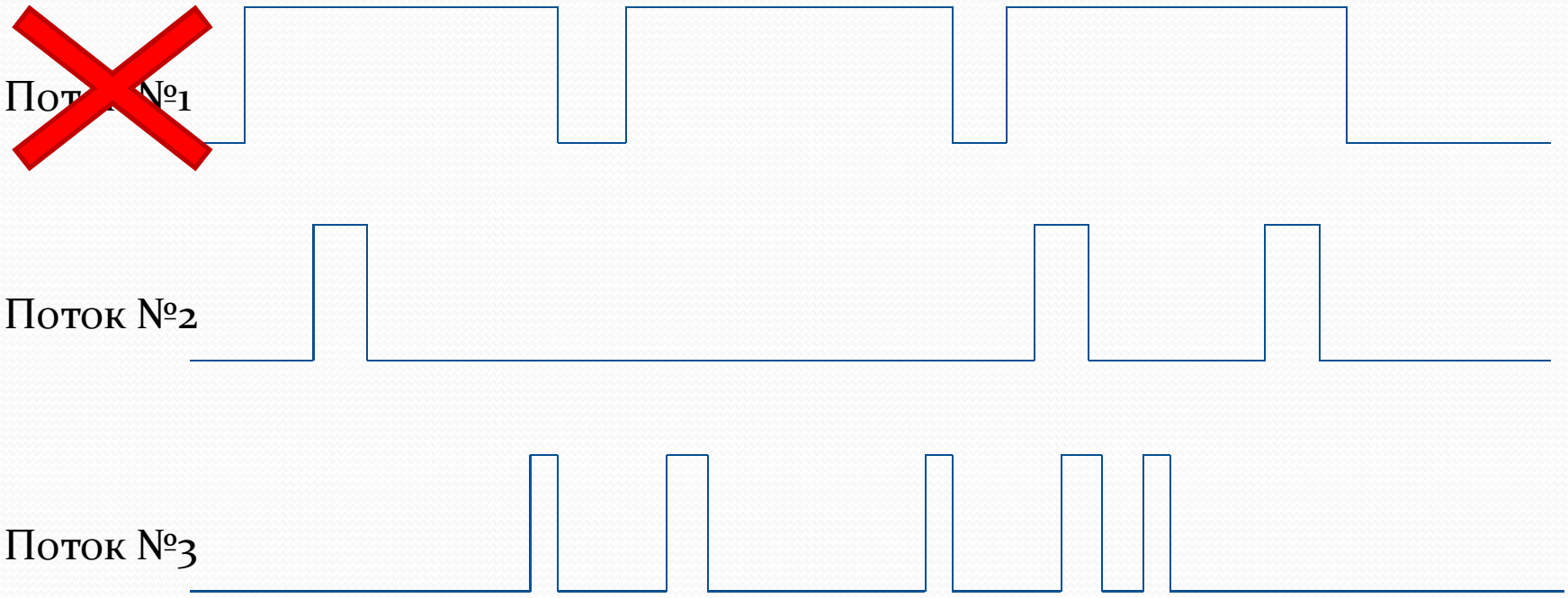
- std::shared_timed_mutex – комбинация shared_mutex + timed_mutex

Проблемы использования

```
volatile int g_counter = 0;                                                    1
std::mutex g_mutex;                                                            3
void func1()                                                                   3
{                                                                              5
       for (int i = 0; i < 100; ++i) {                                         6
            g_mutex.lock();                                                    7
            g_counter++;                                                       8
            std::this_thread::sleep_for(std::chrono::milliseconds(100));       9
            g_mutex.unlock();                                                  10
       }                                                                       11
}                                                                              12
void func2()                                                                   13
{                                                                              14
       for (int i = 0; i < 100; ++i) {                                         15
            g_mutex.lock();                                                    16
            g_counter--;                                                       17
            std::this_thread::sleep_for(std::chrono::milliseconds(100));       18
            g_mutex.unlock();                                                  19
       }                                                                       20
}                                                                              21
void main(void)                                                                22
{                                                                              23
       std::thread t1(func1);                                                  24
       std::thread t2(func2);                                                  25
       for (int i = 0; i < 100; ++i) {                                         26
            std::this_thread::sleep_for(std::chrono::milliseconds(100));       27
            std::cout << g_counter << '\n';                                    28
       }                                                                       29
}                                                                              30
```

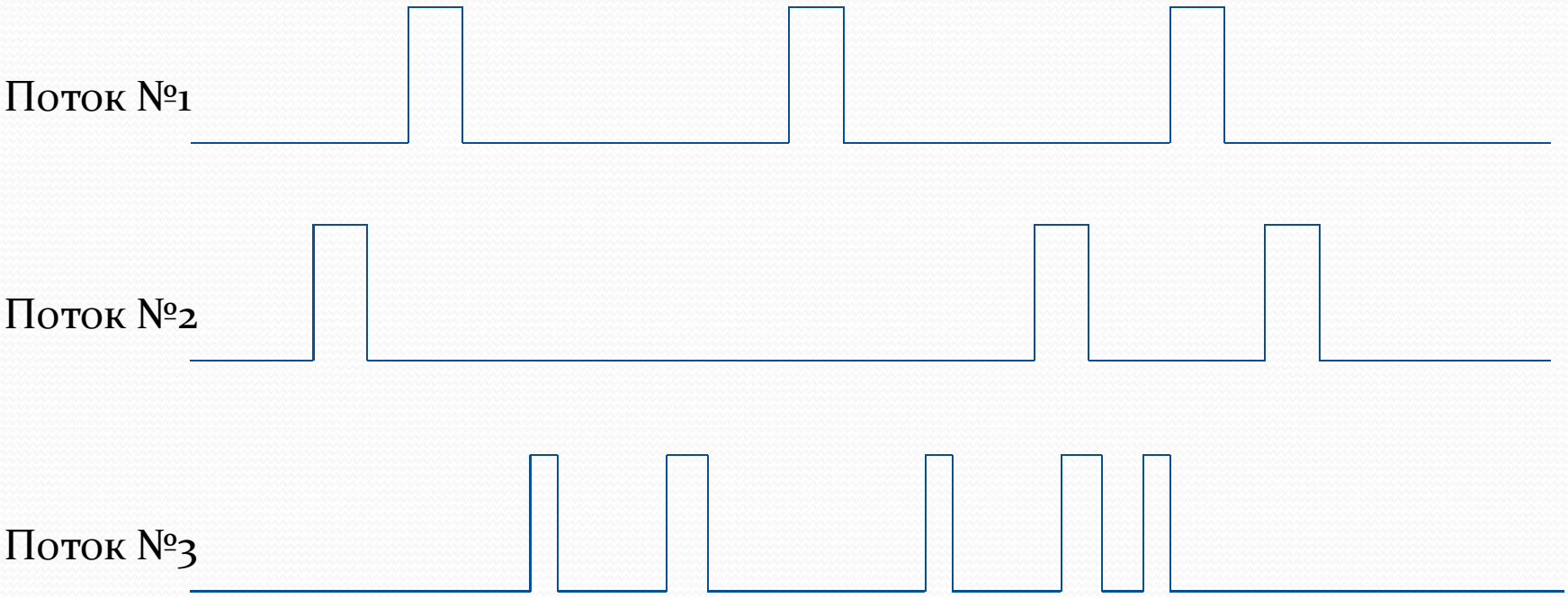
Проблемы использования

- Скважность захвата объекта mutex << 100%



Проблемы использования

- Скважность захвата объекта mutex << 100%



Удобство использования

- std::lock_guard – автоматический захват и освобождение объекта:

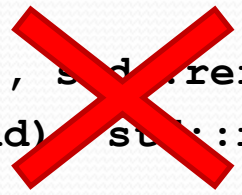
```
{
  std::lock_guard<std::timed_mutex> guard(g_MyMutex);
  g_MyData = …;
}
```

- std::unique_lock – больше функций
  - «умный» конструктор
  - std::lock(l1, l2, …) – множественный захват

Захват нескольких объектов

```
auto call = [](std::mutex& m1, std::mutex& m2)
{
        m1.lock();//#1
        m2.lock();//#2
        …
        m1.unlock();
        m2.unlock();
};

std::mutex first;
std::mutex second;
std::thread firstThred(call, std::ref(first), std::ref(second));
std::thread secondThred(call, std::ref(second), std::ref(first));
```



Захват нескольких объектов

```
auto call = [](std::mutex& m1, std::mutex& m2)
{
        std::lock(m1, m2);
        …
        …
        m1.unlock();
        m2.unlock();
};

std::mutex first;
std::mutex second;
std::thread firstThred(call, std::ref(first), std::ref(second));
std::thread secondThred(call, std::ref(second), std::ref(first));
```

std::condition_variable

- условная переменная
  (передача сигналов между потоками)
- wait / wait_for / wait_until – ждать сигнала
- notify_one – послать сигнал ждущему потоку (одному)
- notify_all – послать сигнал всем ждущим потокам

std::condition_variable

- Фальшивое пробуждение (spurious failure) !!!
  Проверять обычную переменную

```
std::mutex g_m;
std::condition_variable g_cv;
bool g_ready = false;
void worker_thread() {
    // wait until main send data
    std::unique_lock<std::mutex> lk(g_m);
    g_cv.wait(lk, []{return g_ready;});
}
void main() {
    // send data
    {
        std::lock_guard<std::mutex> lk(g_m);
        g_ready = true;
    }
    g_cv.notify_one();
}
```

Атомарные операции

- std::atomic<T> – класс для атомарных операций
  - is_lock_free – true, если для данного типа блокировки не будет.
  - store – Кладет новое значение в объект.
  - load – Извлекает значение из объекта.
  - exchange – Заменяет значение в объекте на новое и возвращает старое.
  - compare_exchange_*(object, expected, desired, success, failure)
    Если object равен expected, тогда desired помещается в object.
    В противном случае object помещается в expected.
    - compare_exchange_weak – compare_exchange с фальшивым пробуждением (spurious failure) – использовать в цикле.
    - compare_exchange_strong – гарантированно возвращает верный результат и не зависит от фальшивой ошибки.

Атомарные операции

- std::atomic<flag>
  - std::atomic_flag_test_and_set / …_explicit – возвращает, что было и записывает true.
  - atomic_flag_clear / …_explicit – записывает false.

- Всегда работает без блокировки !!!

Атомарные операции

- std::atomic<целое>
  - fetch_add(object, value) – атомарно помещает (object + value) в object.
  - fetch_sub(object, value) – атомарно помещает (object – value) в object.
  - fetch_and(object, value) – атомарно помещает (object & value) в object.
  - fetch_or(object, value) – атомарно помещает (object | value) в object.
  - fetch_xor(object, value) – атомарно помещает (object ^ value) в object.

Атомарные операции

- std::atomic<указатель>
  - fetch_add(object, value) – атомарно помещает (object + value) в object.
  - fetch_sub(object, value) – атомарно помещает (object – value) в object.

Атомарные операции

- Чтение-модификация-запись
- Глобальная синхронизация изменения атомарных переменных – порядок изменения:

```
typedef enum memory_order {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst  (Sequentially-consistent ordering =
                           последовательная согласованность)
} memory_order;
```

Атомарные операции

- Чтение-модификация-запись
- Синхронизация потоков по порядку изменения

```
std::atomic_int atomV{0};
int simpleV= 0;

void thread1()
{
    simpleV = 3;
    atomV.store(10);
}

void thread2()
{
    while(atomV.load() != 10);
    assert(simpleV == 3);
}
```

volatile vs atomic<>

- volatile – запрет кэширования в регистре

```
volatile int n = 0;
n++; // без гарантии транзакционности
```

- atomic<int> – гарантированная транзация «чтение-модификация-запись»

```
std::atomic_int n(0);
n++; // гарантируется транзакционность
```

volatile для обмена данными

- volatile – запрет кэширования в регистре

```
volatile bool g_bCanWork = true;
…
while( g_bCanWork ) {
  … // любой код
}
…
g_bCanWork = false;
wait... // ждем завершения работы потока
```

std::future

- Однократное уведомление
- Две части:
  - Флаг готовности
  - Результирующее значение
- Можно передать исключение (try / except)
- Исключительный доступ к результату, который не может быть испорчен кем-то другим

std::promise

```
std::promise<int> p;
std::future<int> f = p.get_future();
std::thread(
      [](std::promise<int>& p){ p.set_value(9); },
      std::ref(p)
).detach();
...
f.wait();
...
int result = f.get();
```

std::async

```
std::future<int> f = std::async(
        std::launch::async,
        [](){ return 8; }
);
...
f.wait();
...
int result = f.get();
```

std::packed_task

```
std::packaged_task<int()> task([](){ return 7; });
std::future<int> f = task.get_future();
std::thread( std::move(task) ).detach();
...
f.wait();
...
int result = f.get();
```

Что использовать?

- std::promise – собственная реализация потоков
- std::async – запуск готовых задач, которые эффективнее исполнять в отдельном потоке, причем запуск этих задач нет смысла откладывать на потом.
- std::packaged_task – просто небольшие задачи, которые не эффективно запускать в отдельном потоке.

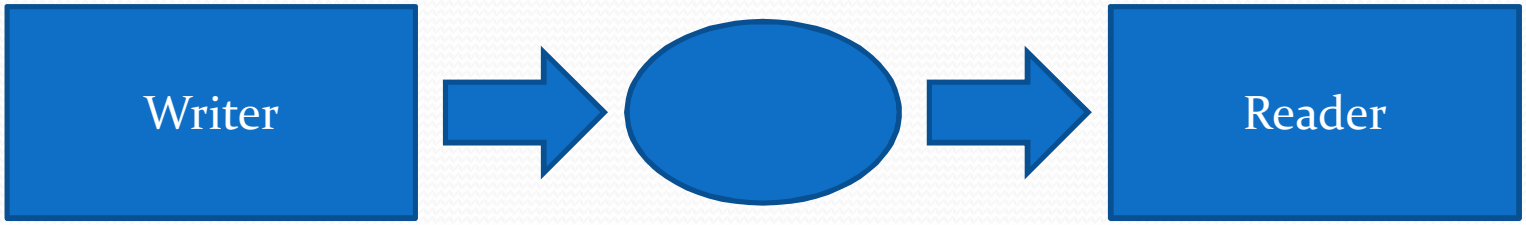
Если ожидаемое время выполнения задачи меньше миллисекунды, то лучше использовать std::packaged_task

Exception from std::future

```
auto f = std::async(
    []() { throw std::bad_alloc(); }
);
...
try {
    f.get();
}
catch(std::exception&) {
    std::cout << "Catch exception !!!\n";
}
```

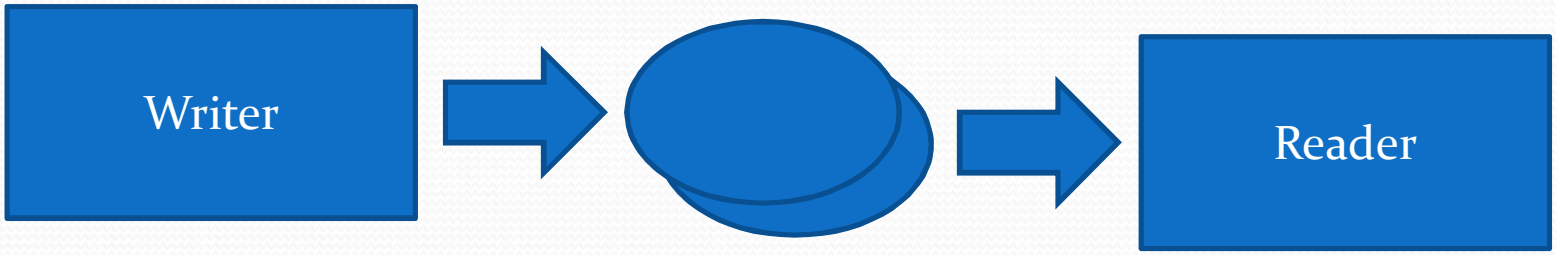
Передача данных

- Писатель генерирует данные
- Читатель потребляет данные
- Есть промежуточный буфер с данными
- Проблема целостности данных
  при одновременной работе писателя и читателя



Flip-flop buffer

- Очень простая реализация – вместо одного блока данных есть два блока (плюс один индекс)
- Читатель гораздо быстрее писателя
- Ситуацией, когда для читателя данные не валидные, можно пренебречь



Flip-flop buffer

```
MyData data[2];
volatile int nReadyIndex = 0;

// writer
int nIndex = nReadyIndex==0 ? 1 : 0;
MyData *ptr = data[nIndex];
… // fill data (ptr)
nReadyIndex = nIndex;

// reader
MyData *ptr = data[nReadyIndex];
… // use data (ptr)
```

FIFO – First Input First Output

- Сглаживание неравномерности обработки данных

- Очень важна средняя скорость каждого из потоков – кто следит за заполненостью FIFO

FIFO – общий вид интерфейса

```
class CFifo {
public:
        // for Writer
        void* GetFree();
        void AddReady(void*);
        // for Reader
        void* GetReady();
        void AddFree(void*);
}
```

FIFO – работа писателя

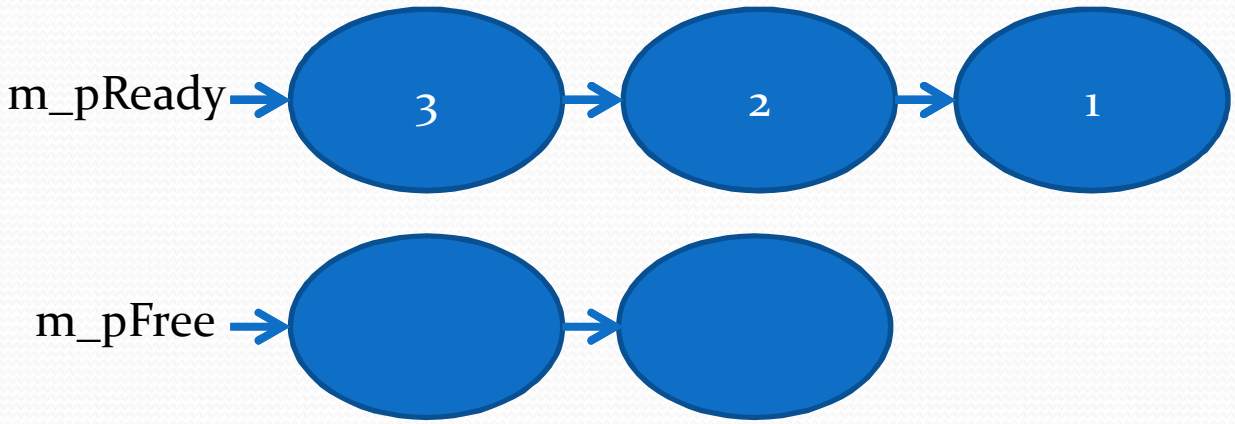
```
while( m_bCanWork ) {
   ...
   void *data = fifo.GetFree();
   if( nullptr != data ) {
      ... // fill data
      fifo.AddReady(data);
   }
}
```

FIFO – работа читателя

```
while( m_bCanWork ) {
   ...
   void *data = fifo.GetReady();
   if( nullptr != data ) {
      ... // use data
      fifo.AddFree(data);
   }
}
```

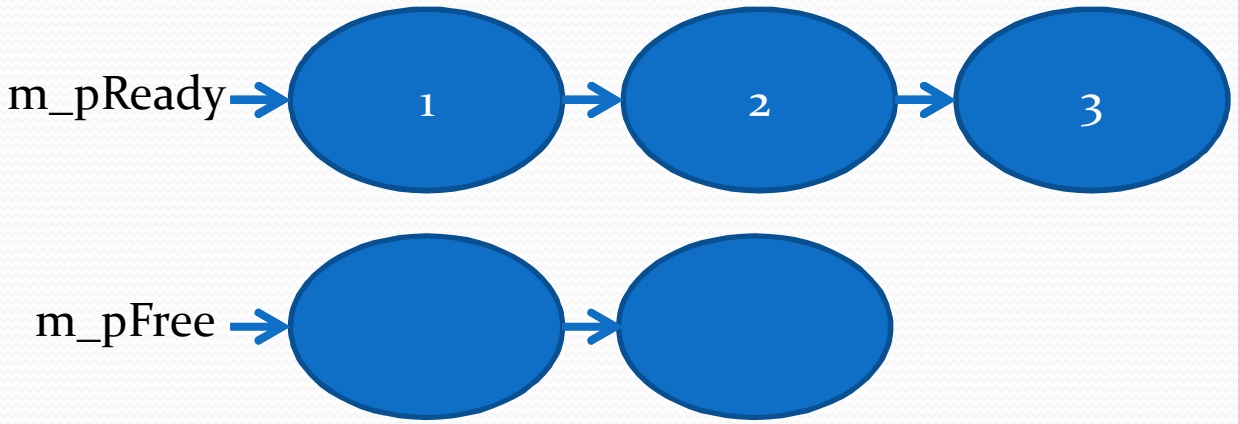
FIFO на списке блоков

- Все данные одного «большого» размера
- Количество данных и размер данных задаются в конструкторе:
  `CFixedFIFO(int nDataSize, int nDataCnt)`



FIFO на списке блоков

- Все данные одного «большого» размера
- Количество данных и размер данных задаются в конструкторе:
  `CFixedFIFO(int nDataSize, int nDataCnt)`



FIFO на списке блоков

```
CFifo::CFifo(size_t data_count, size_t data_size) {
  std::lock_guard<std::mutex> guard(m_FifoMutex);
  for (size_t i=0; i<data_count; ++i)
    m_FreeData.push_pop(std::malloc(data_size));
}

void *CFifo::AddReady(void *data) {
  std::lock_guard<std::mutex> guard(m_FifoMutex);
  m_ReadyData.push_back(data);
}

void *CFifo::GetReady() {
  std::lock_guard<std::mutex> guard(m_FifoMutex);
  void *ptr = m_ReadyData.front();
  if (nullptr != ptr) m_ReadyData.pop_front();
  return ptr;
}
```

FIFO на списке блоков

Варианты:

- Повторное использование готовых данных
- Защита от ошибок программиста – повторный вызов Get… без вызова Add…
- Защита от непрерывного вызова в цикле без ожидания (скважность mutex 100%)
- Копирование данных

FIFO на списке блоков

Варианты:

- Повторное использование готовых данных
- Защита от ошибок программиста – повторный вызов Get… без вызова Add…
- Защита от непрерывного вызова в цикле без ожидания (скважность mutex 100%)
- Копирование данных

FIFO на кольцевом буфере

- Все данные одного маленького размера (размер буфера кратен размеру данных)
- Данных очень много (одновременно пишется и/или читается много данных)

FIFO на кольцевом буфере

```
class CFifo {
public:
        // for Writer
        void* GetFree(size_t data_cnt);
        void AddReady(size_t data_cnt);
        // for Reader
        void* GetReady(size_t &data_cnt);
        void AddFree(size_t data_cnt);
}
```

FIFO на кольцевом буфере

- Все данные одного маленького размера (размер буфера кратен размеру данных)
- Данных очень много (одновременно пишется и/или читается много данных)

m_nReadySize = m_nFree – m_nReady

m_nFreeSize = m_nReady – m_nFree

FIFO на кольцевом буфере

```
void* CFifo::GetFree(size_t data_cnt) {
  if (data_cnt > m_nFreeSize)  return nullptr;
  else  return &m_pData[m_nFree];
}
void CFifo::AddReady(size_t data_cnt) {
  std::lock_guard<std::mutex> guard(m_FifoMutex);
  m_nFree = (m_nFree + data_cnt) % m_nSize;
  m_nFreeSize -= data_cnt;
  m_nReadySize += data_cnt;
}
void* CFifo::GetReady(size_t &data_cnt) {
  if (data_cnt > m_nReadySize)  return nullptr;
  else  return &m_pData[m_nReady];
}
void CFifo::AddFree(size_t data_cnt) {
  std::lock_guard<std::mutex> guard(m_FifoMutex);
  m_nReady = (m_nReady + data_cnt) % m_nSize;
  m_nFreeSize += data_cnt;
  m_nReadySize -= data_cnt;
}
```

Комбинированное FIFO

- Данные переменного размера
- Данные в кольцевом буфере +
  список заголовков (начало и размер данных)
- Что делать, если данные в конце буфера переходят через край?

Межпроцессорное взаимодействие

- Shared memory
- ФИФО на кольцевом буфере
- Комбинированное фифо без указателей
- Синхронизация через системные средства (именованный Event)
- Синхронизация через атомарные операции (без системных средств, только процессор)
- Синхронизация через чтение дубля данных (защитные интервалы, кэш-блоки)