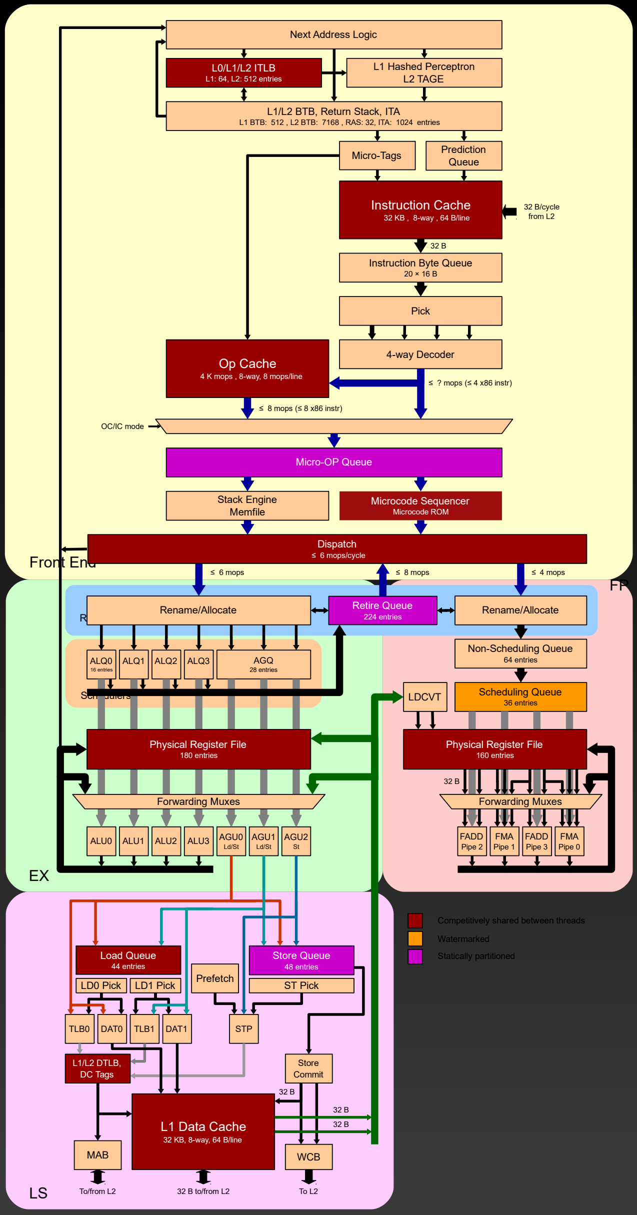
Concurrent vs. parallel vs. distributed

- Concurrent
  - different parts of the program running in single system at the same time, can communicate
- Parallel
  - single computation divided to smaller and same parts, running concurrently
- Distributed
  - concurrent execution on different computers

HW parallelism

- Registers
  - SSE = 128 bits
  - parts of 2^n bytes
- Instructions
  - more ALUs
  - Out Of Order (OOO) execution
- SMT = Simultaneous Multi-Threading
  - more threads (2-16) in single core
  - better ALU utilization
- SMP – Symmetric Multi-Processing
- Cluster – computers + FAST interconnect
- Grid, Cloud – computers + common network



Shared data

- OK only if single process (thread) is changing data
- Locking – necessary if more than one writer
- Locks
  - mutex, critical section, synchronized methods, semaphor, …

Implicit vs. explicit parallelism

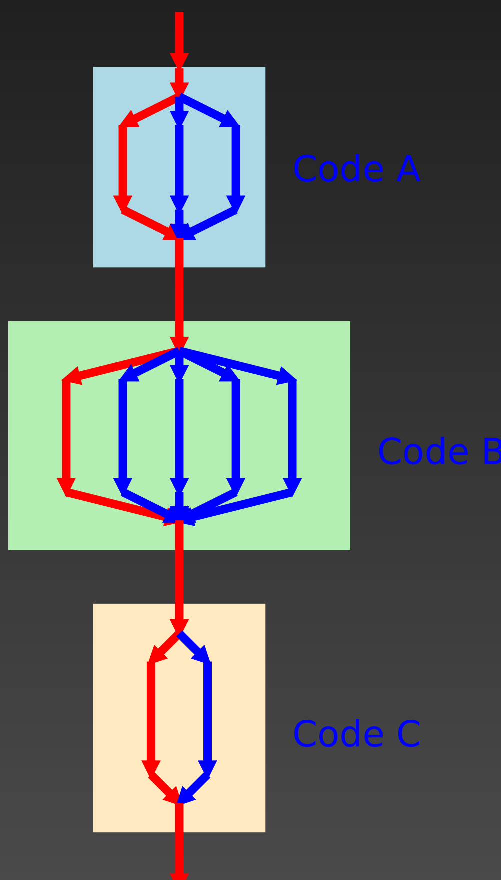
- Implicit
  - Automatic parallelisation by compiler
    - both instruction level and pieces of source code (usually for loops)
  - Precompiled libraries (OpenCV)
  - Implicitly parallel programming languages
    - LabView, Matlab (1:N)
  - No effort to splitting, comm, sync
  - Smaller control over runtime, smaller efficiency, overhead is hidden

Implicit vs. explicit parallelism

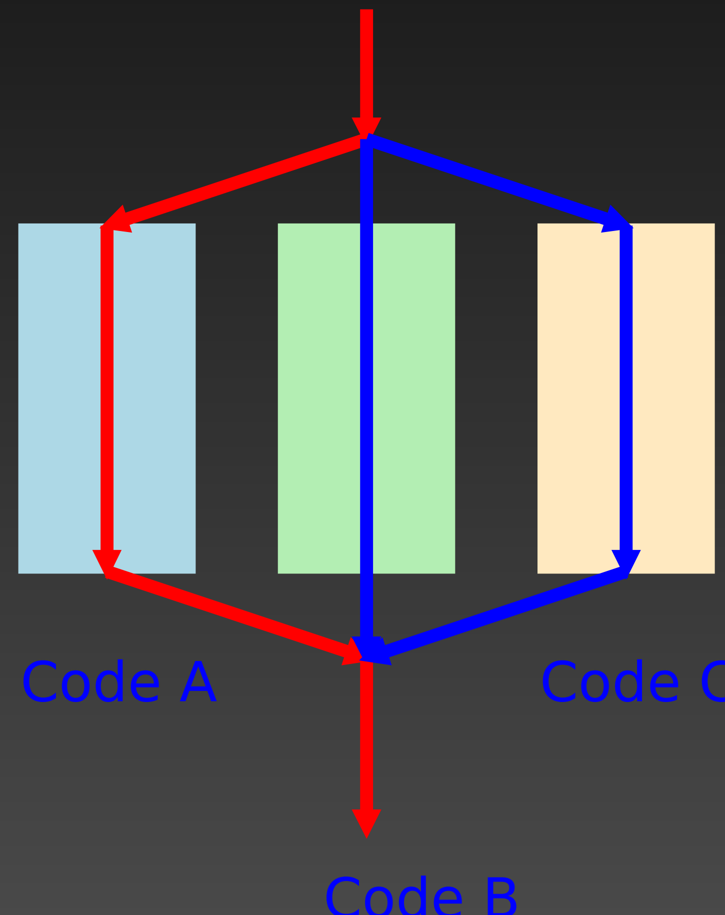
- Explicit
  - Precise control of concurrency, sync, comm using compiler directives, function calls etc.
    - overhead is visible and controllable
  - Directives
    - code splitting
    - synchronisation
    - communication
  - Full controll, higher efficiency possible
  - thread API, OpenMP, MPI

Data vs Task Parallelism

- Data-parallel
  - data are distributed
  - thread code (nearly) same



- Task-parallel
  - code is distributed



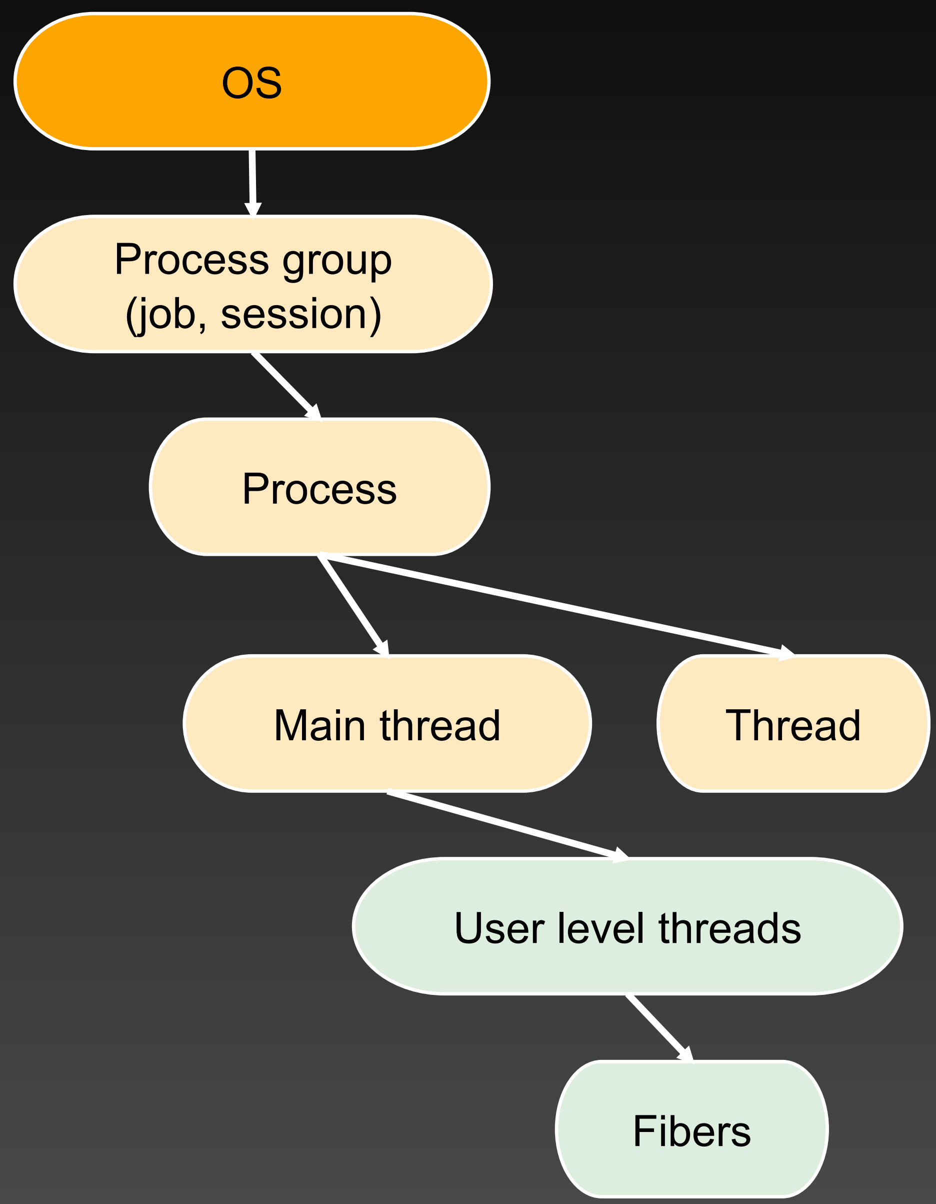
Splitting APP to threads

- Usually data and task parallelism
- Design patterns
  - Master/Slave + thread pool
    - master thread scatter and gather data + control others, does NOT compute itself
    - workers(slaves) threads usually created in advance
  - Equal threads
    - master also works
  - Pipeline
    - task parallel, each threads does different task
    - problems if one stage is slower

Types of parallelism

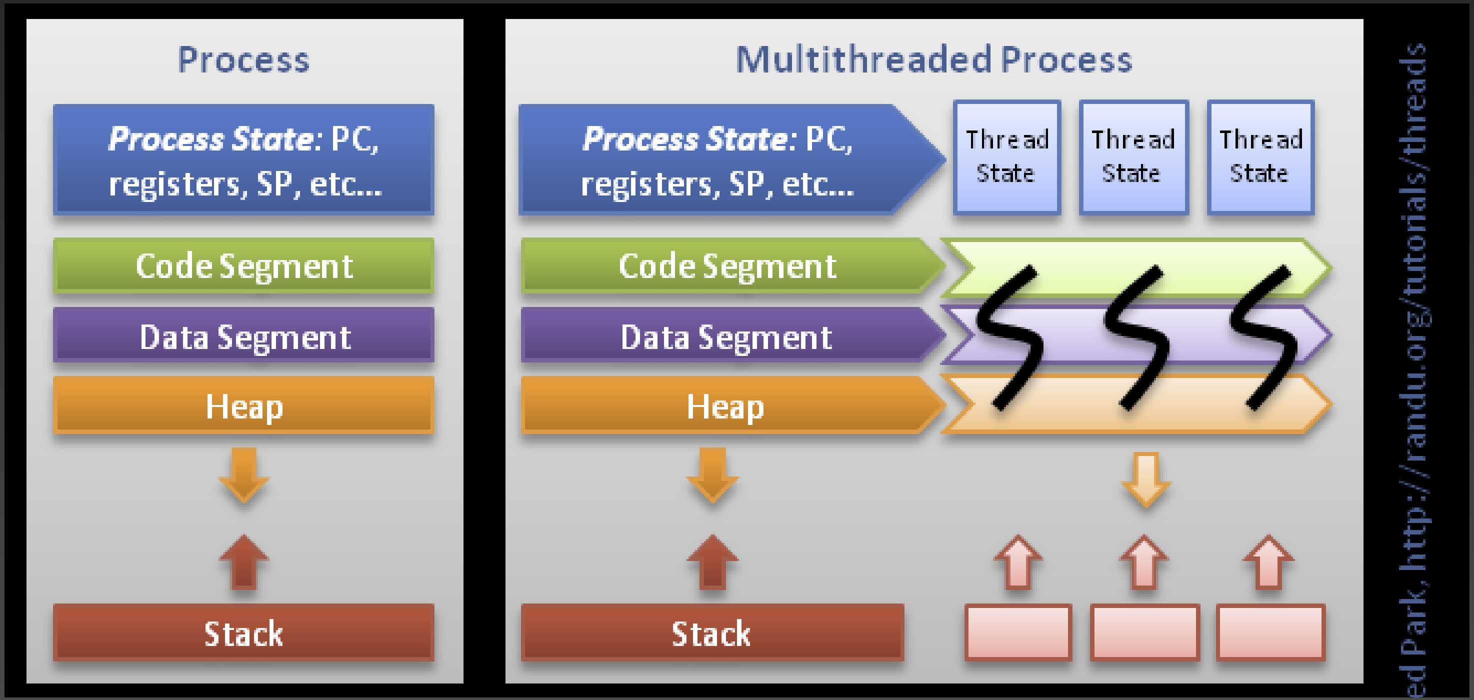
- Fine grained
  - frequent comm and sync
  - small data blocks after shot execution
  - very latency sensitive
- Coarse grained
  - occasional communication
  - sync necessary, but not latency sensitive
- Embarrassingly parallel
  - completely independent tasks, zero comm
  - e.g. repeated run with different cmd line args

Task Hierarchy



Process vs. thread

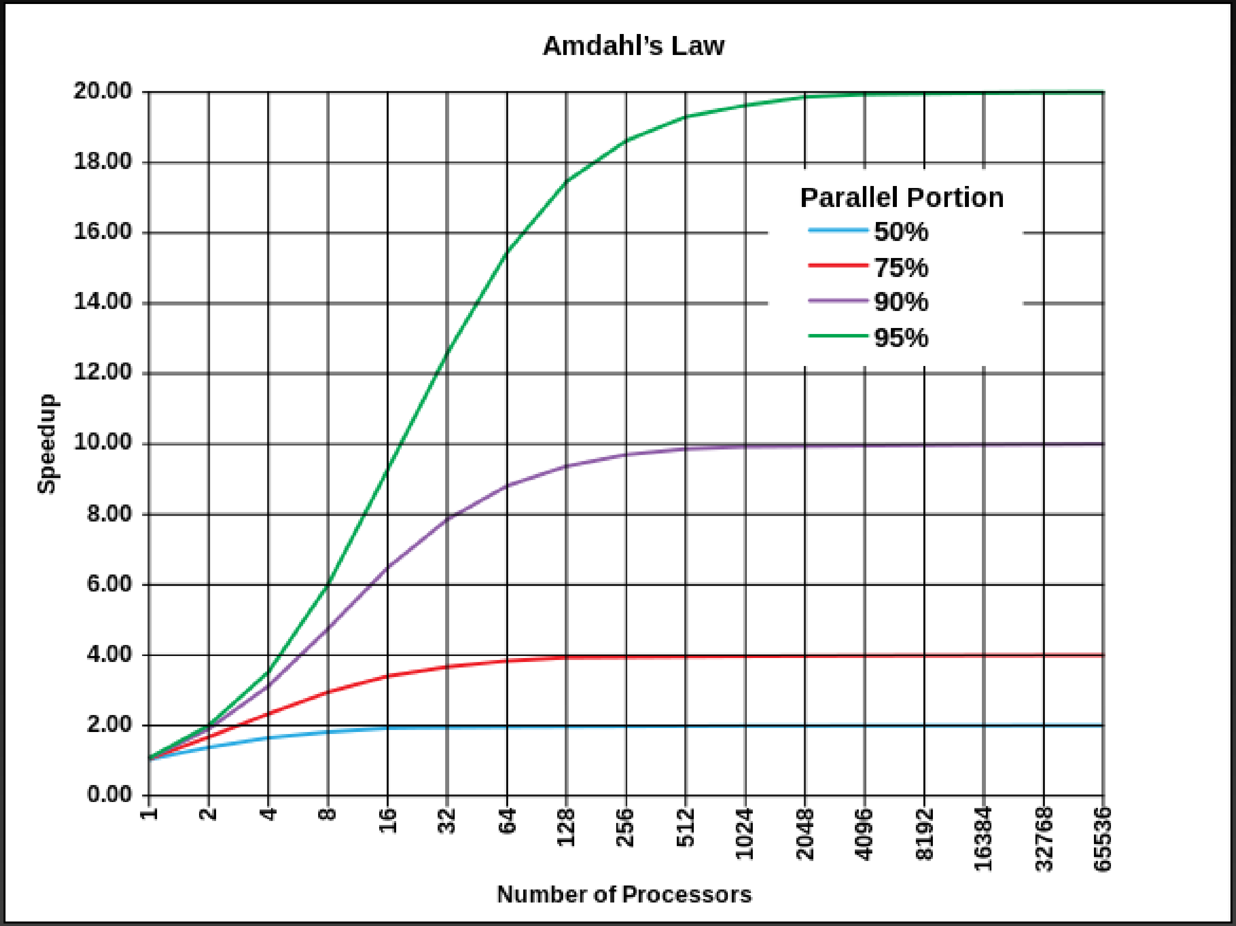
- Process
  - Virtual memory, privileges, code, PID, priority
  - at least one thread
- Thread
  - memory is shared for all threads
  - thread local: only stack, registers, thread ID



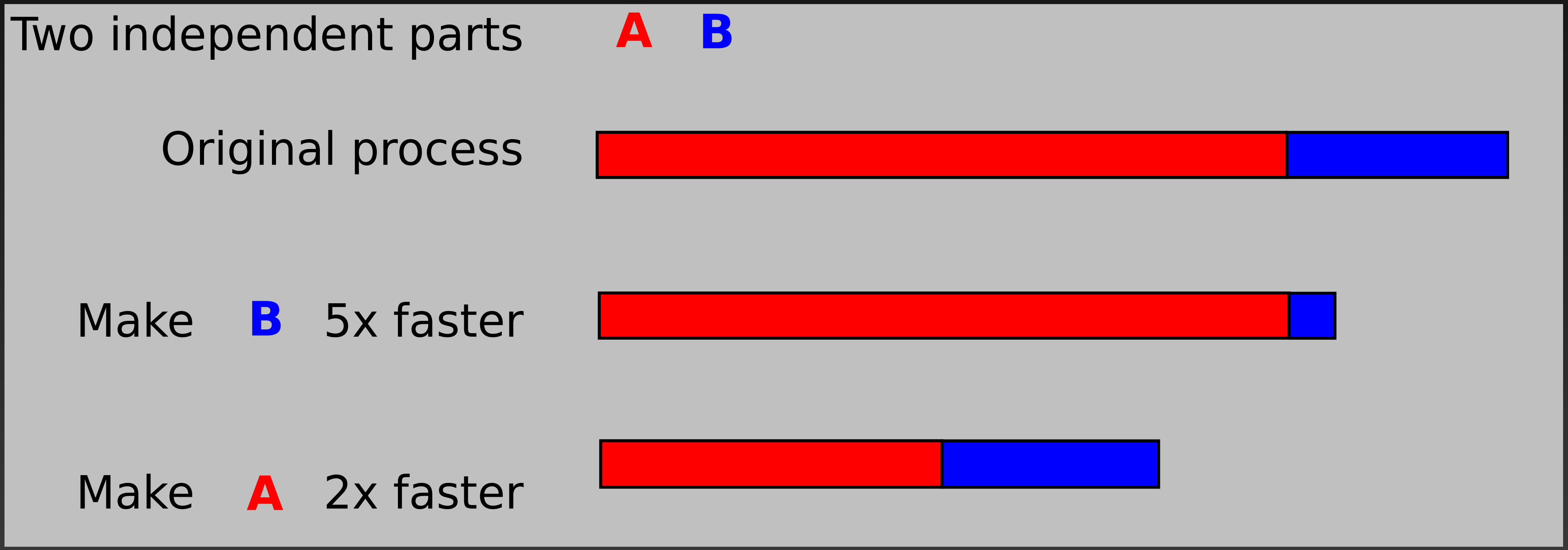
Synchronisation

- MUTEX = MUTual EXclusion
  - lock used for serialisation of thread access to resources
- Critical section
  - code between mutex locking and unlocking
  - guaranteed to be executed by only single thread at once
    - serial time
  - must be as small as possible
- Atomic operations
  - simple operations, guaranteed by hardware or library to be correct without explicit mutex (e.g. ++)

Parallel vs. serial time



Serial time impact



Safety

- Dangerous operation considering parallel execution
  - uncontrolled access to globals (variables and heap)
  - saving function state to global variables
  - global resource (de)allocations (files, sockets, …)
  - indirect access to data using pointers and references
  - visible side effects (modifications of volatile variables)
- Safe strategy
  - use only local variables (stack)
  - code depends only on function arguments, value passing
  - all functions and subfunctions are re-entrant

Synchronisation primitives I

- Mutex – lock (locked vs. unlocked)
- Barrier
  - position in code, where execution of a thread is paused until all threads will arrive
- Join (fork-join)
  - gather results and exit status from all threads – join will terminate thread

Synchronisation primitives II

- Conditional variable
  - call wait() for variable → thread put to sleep
  - HW watching for write into variable → wake up
  - have to check for wake-up reason

```
lock( mutex_x );
while ( not wake_me ) { sleep(cond_var, mutex_x); }
unlock( mutex_x );
```

- Semaphore
  - binary = mutex
  - counted – set to N (resources), thread enters → --, thread exits → ++, on zero → wait

FYI

Posix Threads

FYI

POSIX vs. Win32 Threads

- Similar capabilities
  - win32 handle (=32bit int, for everything) vs. strong typing (each object has own data type)
  - POSIX officially only for C
- implementation of pthreads using win32
  - #include <pthread.h>
  - #include <winpthreads.h>

- http://randu.org/tutorials/threads/
- http://locklessinc.com/articles/pthreads_on_windows

FYI

Calls

- Over 60 API functions
- prefix for entity type, suffix for operation
  - pthread_, pthread_attr_
  - pthread_mutex_, pthread_mutexattr_
  - pthread_cond_, pthread_condattr_
  - pthread_key_

FYI

Attributes

- Properties of entity (thread, mutex, cond. variable) is set with special objects – attribute objects
- Some entity properties must be specified before entity creation
- Attribute object types
  - Thread: pthread_attr_t
  - Mutex: pthread_mutexattr_t
  - Conditional variable: pthread_condattr_t
- Creation and destruction
  - function _init(...) and _destroy(...) with prefix
  - parameter set to pointer to attribute object

FYI

Thread creation

- Each program has one main thread, created by OS
- Other threads created explicitly
- Each thread can start more threads
- Thread is created by pthread_create()
- Thread is immediately ready to run
  - It can be started by OS scheduler before parent thread returns from pthread_create() function
  - All data necessary for thread must be prepared BEFORE calling pthread_create()

FYI

Thread creation

- int pthread_create(pthread_t *thread_handle,

  const pthread_attr_t *attribute,

  void * (*thread_function)(void *), void *arg);
- thread_handle – thread descriptor
- attribute – pointer to structure with attributes of created thread (NULL for standard settings)
- thread_function – pointer to function to execute in thread
- arg – pointer to parameters of thread_function
- returns 0 if successful

FYI

Thread properties

- Detached threads
  - Can not be joined with master by pthread_join()
  - Run on background, saves app resources
  - Standard thread properties are not always obvious → explicit setting recommended
    - int pthread_detach (ptrhead_t *thread_handle)
    - int pthread_attr setdetachstate(pthread_attr_t *attr, int detachstate)
    - int pthread_attr getdetachstate(pthread_attr_t *attr, int *detachstate)

FYI

Terminating thread

- Thread can be terminated
  - by calling pthread_exit() from inside
  - by ending parent thread execution by different call than pthread_exit() (e.g. exit(), abort(), return, ...)
  - by cancelling using pthread_cancel()
  - by ending master thread other than return (kill, exit, abort...)
- void pthread_exit (void *value)
  - terminates thread execution
  - process resources (fd, IPC, mutex, …) created (opened) in thread are NOT closed (deallocated) – global resources
  - heap data referenced only from thread must be released before exit – memory leak (system will release all resources on process termination, not thread)
- Pointer is returned after thread join – use e.g. for returning result

FYI

Thread cancelling

- int pthread_cancel (ptrhead_t *thread_handle)
  - Request for thread_handle thread termination
  - Adressed thread may or may not terminate (just request)
  - Thread may cancel itself
  - Cancel request offers opportunity to do clean-up mem/file/etc. related to thread
  - Function exits after sending request – non-blocking
  - Return code 0 means addressed thread exists, not that it was/is/will be terminated

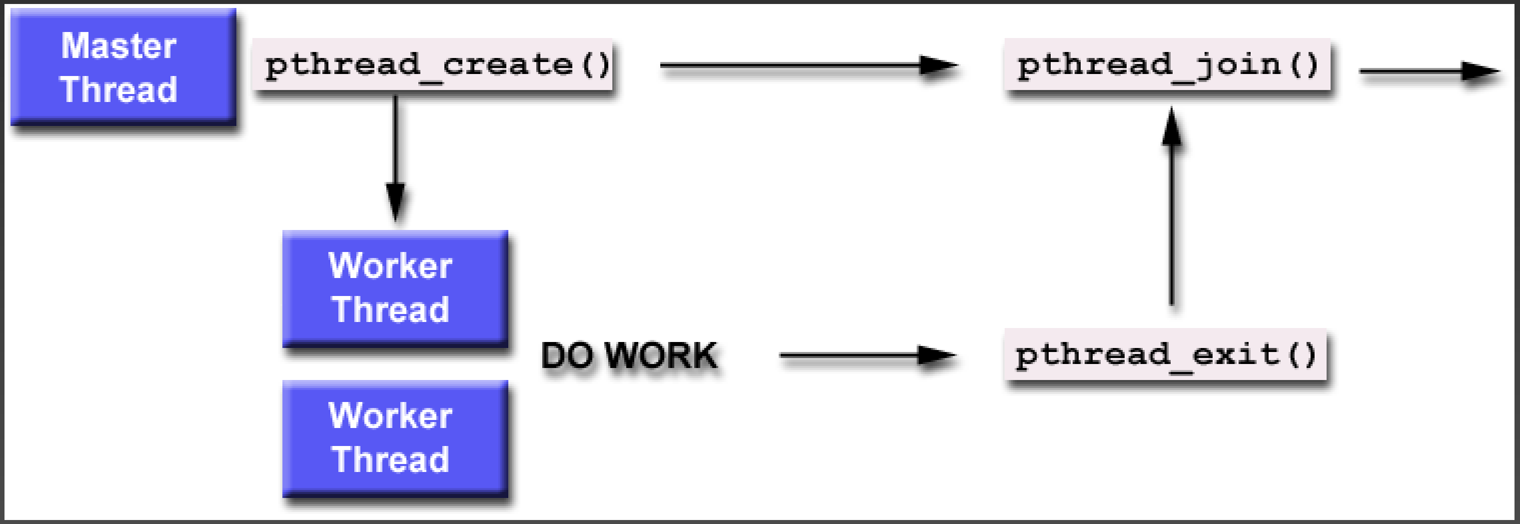
FYI

Thread Join

int pthread_join (pthread_t thread_handle, void **ptr_value);

- blocking wait for thread thread_handle finish
- Value ptr_value is pointer to pointer, specified in thread_handle in pthread_exit()

- Necessary if we want to know exit status code



FYI

Mutex

- Init

  pthread_mutex_t mutex;

  int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

  - NULL attr = default
  - or macro
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
- Usage

  int pthread_mutex_lock(pthread_mutex_t *mutex);

  int pthread_mutex_trylock(pthread_mutex_t *mutex);

  int pthread_mutex_unlock(pthread_mutex_t *mutex);

FYI

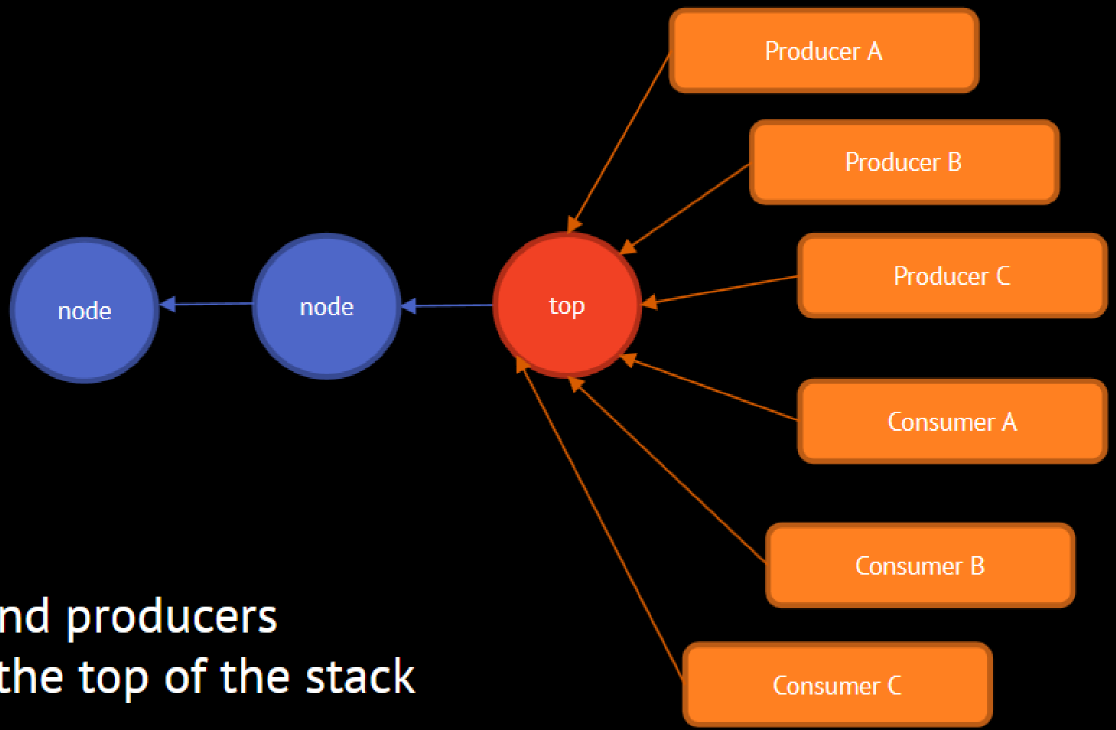
Sample code

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid)
{ printf("%d: Hello World!\n", threadid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{ pthread_t threads[NUM_THREADS];
  for(int t=0; t<NUM_THREADS; t++)
    pthread_create(&threads[t], NULL, PrintHello, (void *)t);
  pthread_exit(NULL);
}
```

Producer – Consumer

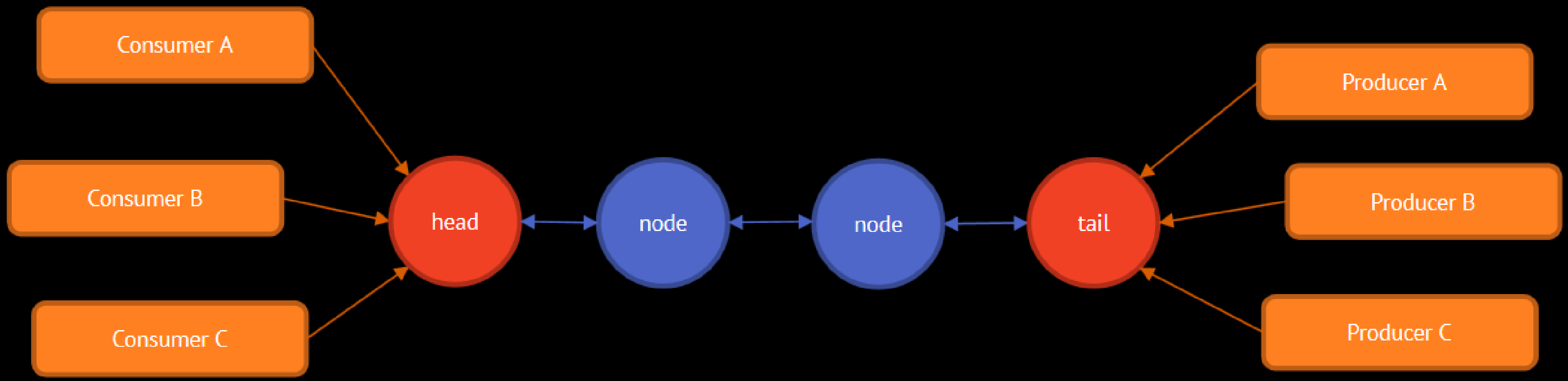
- Thread P produces data, C consumes
- Possible solution
  - shared data storage + mutex
- Better
  - common queue
  - counting semaphore, P increases, C decreases
  - at zero C can be put to sleep
  - problem: more C or P – removing and inserting is not atomic, PxC resource overwriting
- Best
  - conditional variables
  - any amount of C a P, single storage, no busy wait

Data Structures
ConcurrentStack



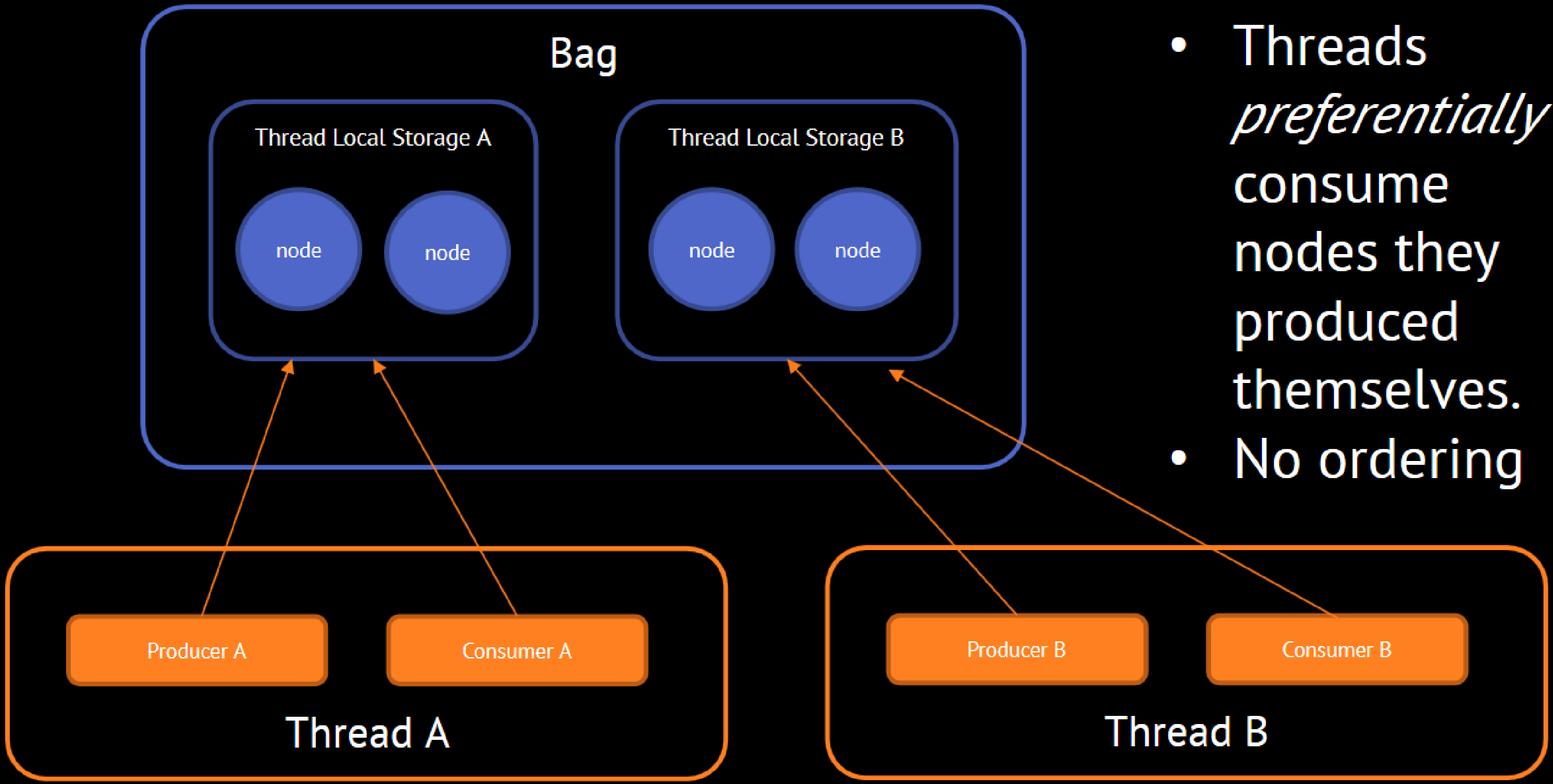
Consumers and producers compete for the top of the stack

Data Structures
ConcurrentQueue



Consumers compete for the head.
Producers compete for the tail.
Both compete for the same node if the queue is empty.

Data Structures
ConcurrentBag



- Threads *preferentially* consume nodes they produced themselves.
- No ordering

@gfraiteur

Producer - Consumer

- Slightly better (one P, more C)
  - more queues with pointers: P and each C has its own Q, one is common
  - single mutex for pointer switching
  - if C empty, lock common, switch for its own, release mutex
    - beware, busy-wait, live-lock!
  - P after each piece of data produced: lock mutex, switch common with its own, release

HW Vlákna v C++

- knihovna <thread>
- kolik máme v CPU hw vláken? (včetně SMT)

```
#include <iostream>
#include <thread>
#include <chrono>

int main(int argc, char** argv) {
    const unsigned int hw = std::thread::hardware_concurrency();

    // Do I care, what data type it is? NO! Auto-deduce type...
    auto hw2 = std::thread::hardware_concurrency();

    std::cout << "Counting threads...\n";                    // End-Of-Line without flush (faster)
    std::this_thread::sleep_for(std::chrono::milliseconds(25));
    std::this_thread::sleep_for(std::chrono::seconds(3));
    std::cout << "Got HW threads: " << hw << std::endl;     // End-Of-Line with implicit flush (safer, slower)

    return EXIT_SUCCESS;
}
```

Fork & join

- vytvoření vlákna vs. skupiny vláken, předání parametru

```cpp
#include <iostream>
#include <thread>

void thread_code(void) {
    std::cout << "Hello world from thread: " <<
std::this_thread::get_id() << "\n";

}

int main(int argc, char** argv) {
    std::thread my_thread(thread_code);

    my_thread.join();

    return EXIT_SUCCESS;
}
```

```cpp
#include <iostream>
#include <vector>
#include <thread>

static const int num_threads = 10;

void thread_code(const int tid) {
    std::cout << tid << std::endl;
}

int main(int argc, char** argv) {
    std::vector<std::thread> threads;

    threads.resize(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        threads[i] = std::thread(thread_code, i);
    }

    for (int i = 0; i < 10; ++i) {
        threads[i].join();
    }

    return EXIT_SUCCESS;
}
```

Rozlišení vláken

- zjistíme hlavní vlákno, podle toho rozhodneme co dělat

```
#include <iostream>
#include <thread>

std::thread::id main_thread_id = std::this_thread::get_id();

void am_i_main(void)
{
    if (main_thread_id == std::this_thread::get_id())
        std::cout << "This is the main thread.\n";
    else
        std::cout << "This is not the main thread.\n";
}

void thread_code(void) {
    std::cout << "Hello world from thread: " << std::this_thread::get_id() << ". ";
    am_i_main();
}

int main(int argc, char** argv) {
    std::thread my_thread(thread_code);

    am_i_main();

    my_thread.join();

    return EXIT_SUCCESS;
}
```

```
This is the main thread.
Hello world from thread: 3832. This is not the main thread.
```

Atomic

- synchronizovaná komunikace mezi vlákny
  - vhodné jen pro malé datové objemy

```
#include <iostream>
#include <vector>
#include <thread>
#include <atomic>

static const int num_threads = 10;

void thread_code(const int tid, std::atomic<int>& result) {
    std::cout << tid << std::endl;

    result += 1;
}

int main(int argc, char** argv) {
    std::vector<std::thread> threads;
    std::atomic<int> result(0);

    threads.resize(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        threads[i] = std::thread(thread_code, i, std::ref(result));
    }

    for (int i = 0; i < 10; ++i) {
        threads[i].join();
    }

    std::cout << "Result: " << result << std::endl;

    return EXIT_SUCCESS;
}
```

Mutex

- synchronizovaná komunikace mezi vlákny
  - v kritické sekci jsou možné i složitější operace

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <thread>
#include <mutex>

static std::mutex my_mutex;

static const int num_threads = 10;

void thread_code(const int tid, int& result) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::thread::id this_id = std::this_thread::get_id();

    // try to move lock BELOW printing
    my_mutex.lock();
    std::cout << "I am " << tid << " with id " << this_id << std::endl;

    result += 1;
    my_mutex.unlock();
}

int main(int argc, char** argv) {
    std::vector<std::thread> threads;
    int result = 0;

    threads.resize(num_threads);
    for (int i = 0; i < num_threads; ++i) {
        threads[i] = std::thread(thread_code, i, std::ref(result));
    }

    for (int i = 0; i < num_threads; ++i) {
        threads[i].join();
    }

    std::cout << "Result: " << result << std::endl;

    return EXIT_SUCCESS;
}
```

• Podmíněné proměnné

umožňují usínání a probouzení vláken

```cpp
#include <iostream>,#include <string>,#include <thread>,#include
<mutex>,#include <condition_variable>

std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread()
{
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, [] {return ready; });

    // after the wait, we own the lock.
    std::cout << "Worker thread is processing data\n";
    data += " after processing";

    // Send data back to main()
    processed = true;
    std::cout << "Worker thr. signals data processing completed\n";

    // Manual unlocking is done before notifying, to avoid waking up
    // the waiting thread only to block again (see notify_one for
details)
    lk.unlock();
    cv.notify_one();
}
```

```cpp
int main()
{
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
        std::cout << "main() signals data ready for processing\n";
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, [] {return processed; });
    }
    std::cout << "Back in main(), data = " << data << '\n';

    worker.join();
}
```

Úprava kontejneru na thread-safe

- přidání zámku pro vynucení exkluzivního přístupu

```cpp
#include <deque>
#include <iostream>          // std::cout
#include <mutex>             // std::mutex, std::scoped_lock
#include <condition_variable> // std::condition_variable

template<typename T>
class synced_deque {
protected:
    std::mutex mux;
    std::deque<T> de_queue;
    std::condition_variable cv_sleep;
    std::mutex mux_sleep;

public:
    synced_deque() = default;
    synced_deque(const synced_deque<T>&) = delete;
    virtual ~synced_deque() {
        clear();
    }

   // Returns and maintains item at front of Queue
    const T& front() {
        std::scoped_lock lock(mux);
        return de_queue.front();
    }

    // Removes and returns item from front of Queue
    T pop_front() {
        std::scoped_lock lock(mux);
        auto t = std::move(de_queue.front());
        de_queue.pop_front();
        return t;
    }
```

```cpp
    // Adds an item to back of Queue
     void push_back(const T& item) {
         std::scoped_lock lock(mux);
         de_queue.emplace_back(std::move(item));

         std::unique_lock<std::mutex> ul(mux_sleep);
         cv_sleep.notify_one();
     }

    // Returns true if Queue has no items
     bool empty() {
         std::scoped_lock lock(mux);
         return de_queue.empty();
     }

    void wait() {
         while (empty()) {
             std::unique_lock<std::mutex> ul(mux_sleep);
             cv_sleep.wait(ul);
         }

    // const T& back();
    // T pop_back()
    // void push_front()
    // const T& at();
    // size_t size();
    // ...

     }
```