Memory Model

Get your shared data under control

Jana Machutová

Why you want to know more about memory model

World is not single threaded anymore

You will need little bit of concurrency everywhere

Even tiny little piece of concurrent code may cause huge problems if not written correctly

Choosing fitting synchronization technique will save you a lot of pain

Synchronization is still evolving and current state may be different from the one you’ve learned during studies

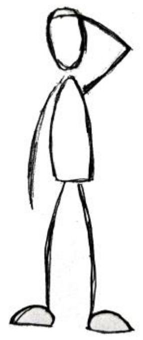
IT’S FUN !!!

Meet Hedvika and Dan

They will help us to better understand synchronization problems on their real world scenario

Hedvika loves hiking

Dan loves it too



Bus to starting point

Walk

Visit medieval castle

Camp sleep over

Walk

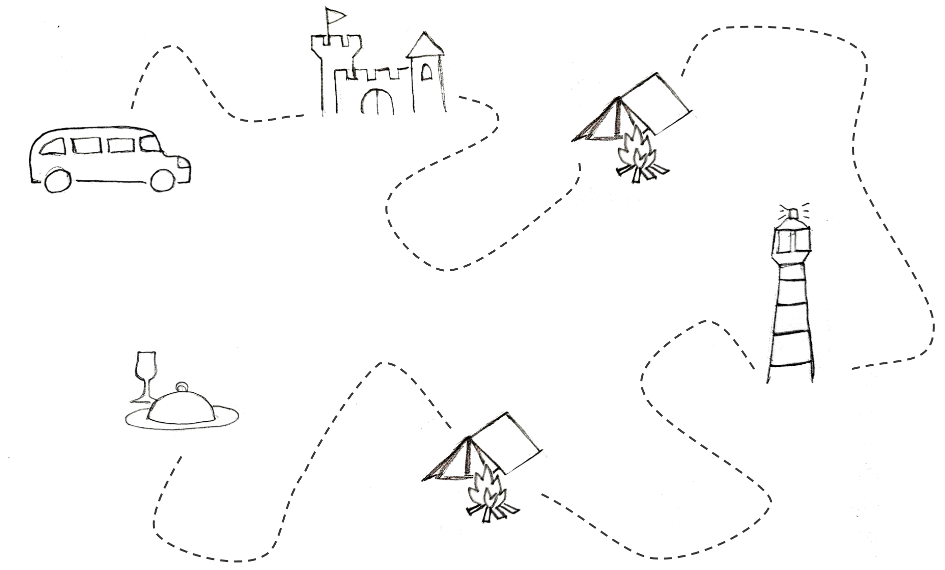
Visit lighthouse

Relax on beach nearby

Second camp sleep over

Walk

Final lunch together



Where is Hedvika?

Without any shared data we don't know anything about other threads

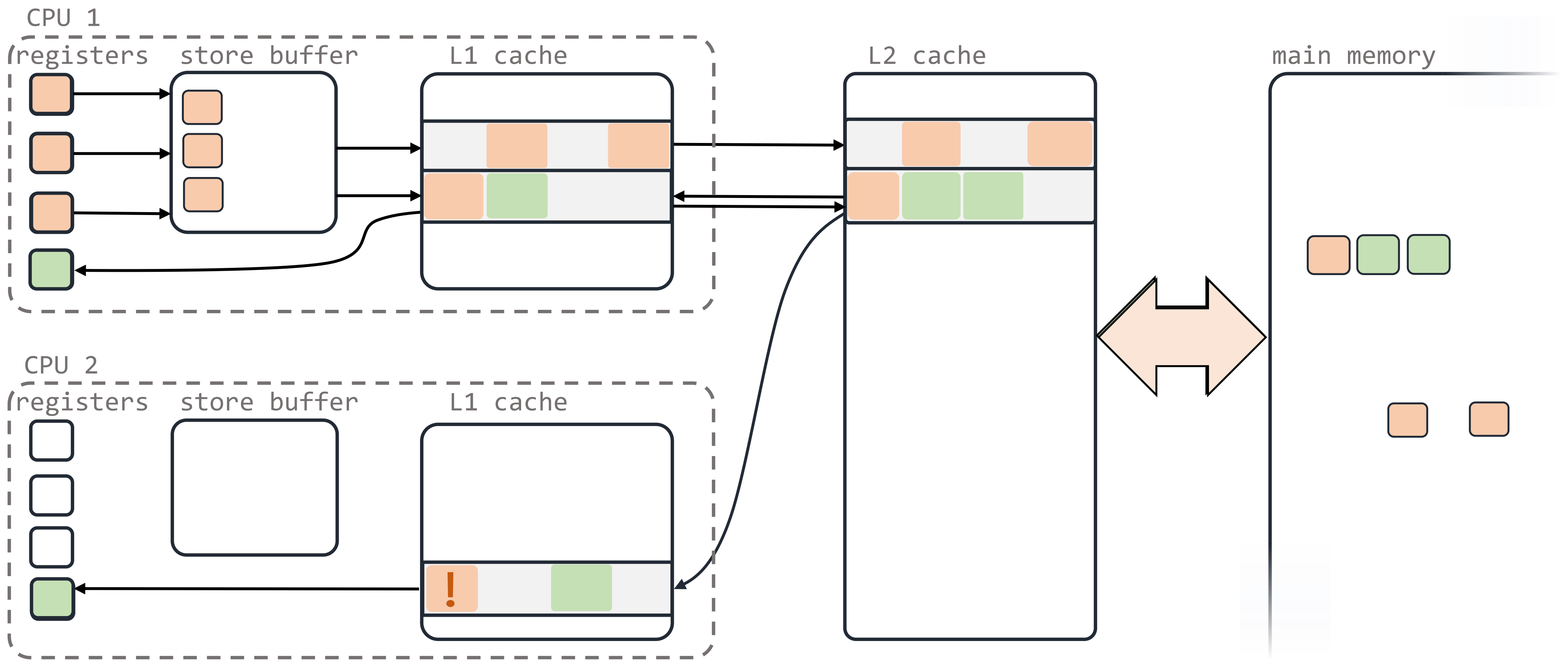
```
//hiking day #2
wakeup();
walk();
photos = visitLighthouse(camera);
publish(photos.GetBest());
beachTime();
sleepInCamp();
```

- Social media as a shared memory
- Hedvika posts picture from the beach
- Dan gets notification about it
- Does he know if she's seen lighthouse already?
- **NO!**
- Hedvika may be really tired after long trip so she rather jump to the sea first
- The same may happen with your code

My code is executed in different order? Why? How? Who've done that?

How data flows through memory hierarchy

To be able to understand possible code reordering we need be familiar with data flow and memory hierarchy

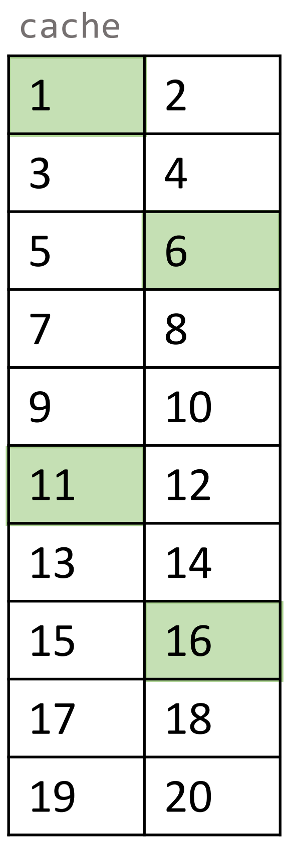


Who changes your code order

Compiler will reorganize your lines during optimization

- **loop reordered to respect cache lines**

```
int buffer[4][5] = {{1, 2, 3, 4, 5},
                    {6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15},
                    {16, 17, 18, 19, 20}};
int twice[4][5];

for(int j = 0; j < 5; ++j)
  for(int i = 0; i < 4; ++i)
    twice[i][j] = buffer[i][j] << 1;
```

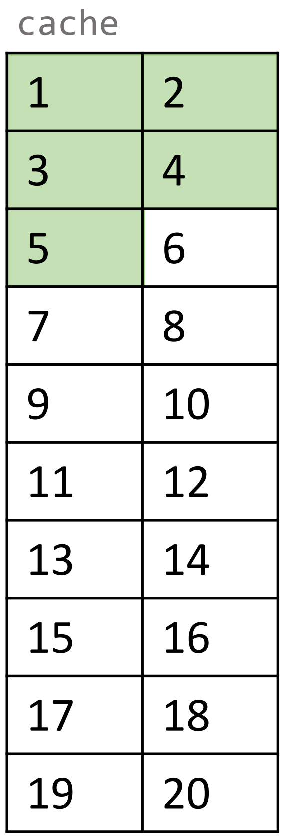


Who changes your code order

Compiler will reorganize your lines during optimization

- **loop reordered to respect cache lines**

```
int buffer[4][5] = {{1, 2, 3, 4, 5},
                    {6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15},
                    {16, 17, 18, 19, 20}};
int twice[4][5];

for(int i = 0; i < 4; ++i)
  for(int j = 0; j < 5; ++j)
    twice[i][j] = buffer[i][j] << 1;
```



Who changes your code order

Compiler will reorganize your lines during optimization

- loop reordered to respect cache lines
- **already loaded variable in registers – put computation using it nearby**

```
int a = 1;
int b = 2;

int c = 3;
int d = 4;
int e = computeStuff(c, d);

++a;
++b;
```

Who changes your code order

Compiler will reorganize your lines during optimization

- loop reordered to respect cache lines
- **already loaded variable in registers – put computation using it nearby**
- and many more…

```
int a = 1;
++a;
int b = 2;
++b;

int c = 3;
int d = 4;
int e = computeStuff(c, d);
```

Who changes your code order

Compiler will reorganize your lines during optimization

- loop reordered to respect cache lines
- already loaded variable in registers – put computation using it nearby
- and many more…

Processor

- dynamic optimizations

Store buffer

- Publishing of newly written value is delayed and may be visible later than you expect

Those are just simple examples for better insight, but there is plenty of other optimizations which can mess up with your code

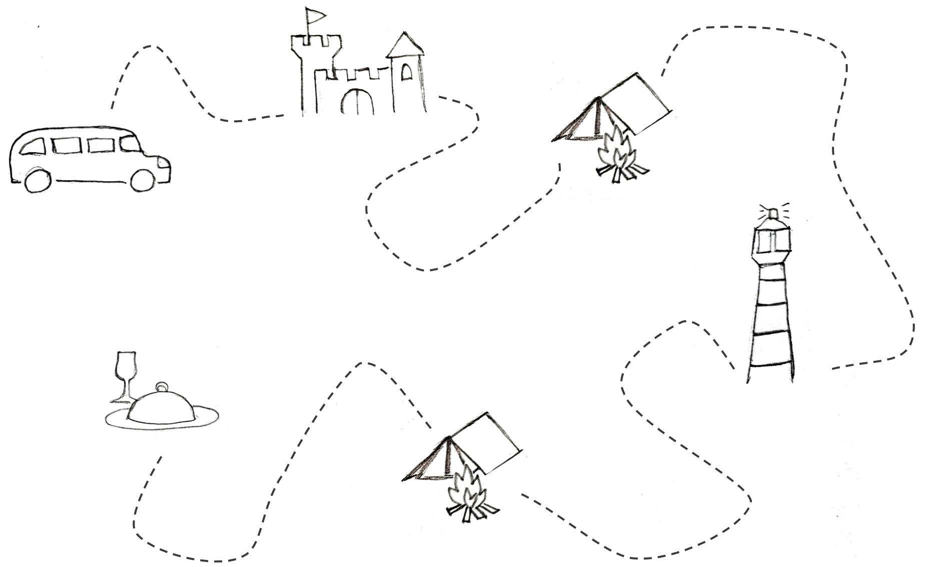
Back to our hike

Hike is getting popular on social media

Hedvika organizes tour for introverts

Everyone is going to enjoy this trip alone, but they need to meet time to time

Hikers are our threads we will try to synchronize at specific points



Critical points requiring communication between hikers

- Get to the same bus
- Get into medieval armor for a nice picture
- Share a tent in small groups of 3
- Enjoy lighthouse with limit of 6 people staying there at one time
- Order and pay the final lunch

Seat everyone in the bus

```
int seatedHikers = 0; //shared counter

//get to the bus – each hiker/thread running this
Food snack = apple;
++seatedHikers;
while (seatedHikers < hikersCount){}
sleepyTraveling();
snack = gasStation.buy(sandwich);
```

What may go wrong?

- Concurrent write to shared variable with unknown result (actually UB)
- Busy waiting drains your CPU
- Danger of code order change
- Buying sandwich may be executed before getting to the bus.
- Compiler will couple two operations on the same variable already loaded in registry

Memory model is here to solve those issues

- Describes the interactions of threads through memory and their manipulation with shared data
- Defines set of rules on how to express our synchronization intentions

- Compilers and processors will understand us and won’t perform dangerous optimizations

- We still want to keep as many optimizations as possible
- Don't share without good reasoning – the performance impact may be huge

- Set of rules is intended to stay minimalistic to give compilers and processors free hands for their work
- Minimalistic rules set leave us with possibility to run into undefined behavior
- If you will follow the rules, you will be safe of UBs

Memory model's behavior

Multiple reads are safe

Single write without any concurrent reads or writes is safe

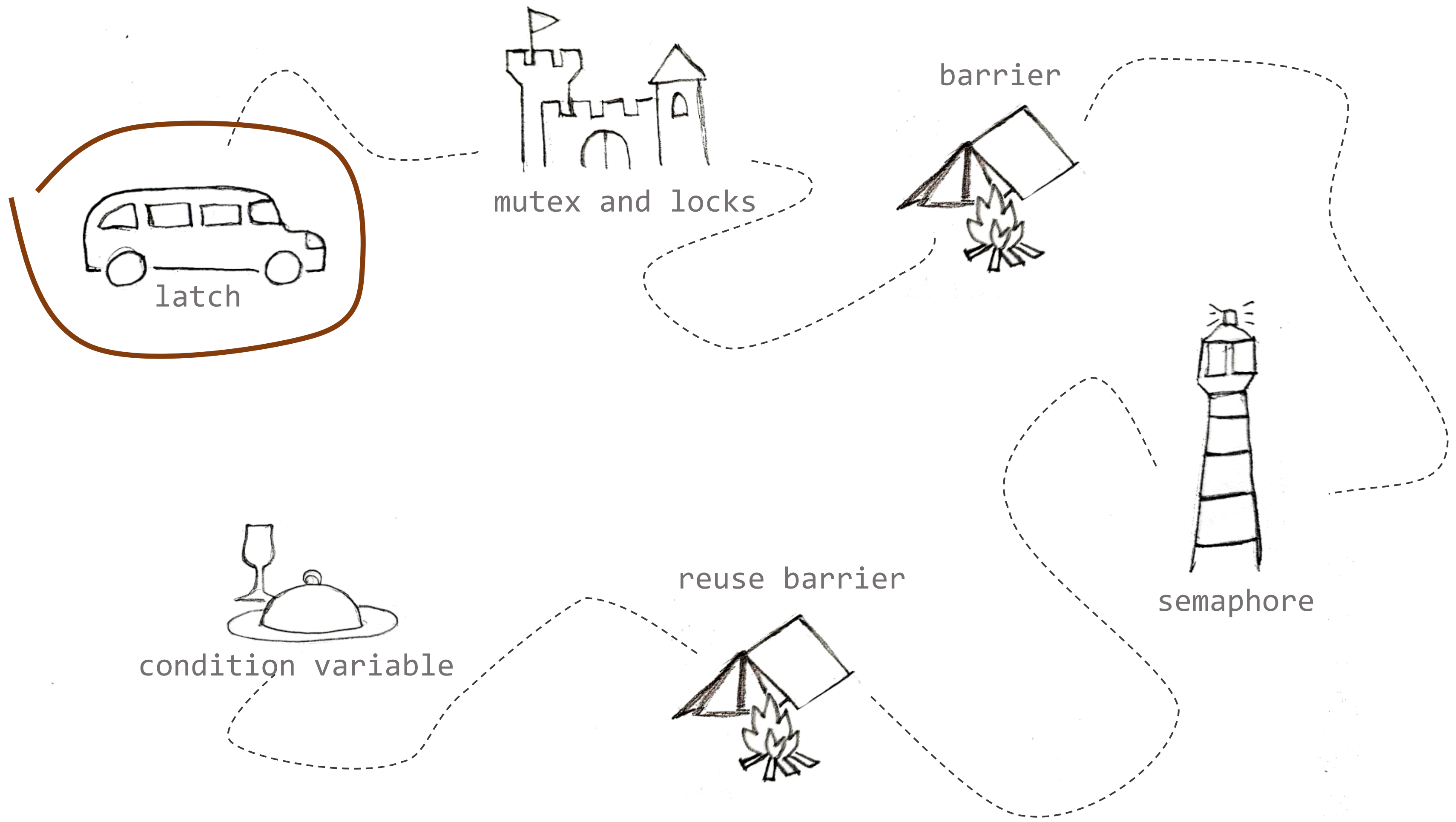
Once you write to memory location and want to read it or write to it again

- You need to perform all of it from **one thread of execution**
- You need to define **happens-before** relation for every read/write
- You always need to access that location **atomically**

Everything else is UB

Or more optimistically - you will be fine if you will follow the rules

Everyone is seated thanks to LATCH

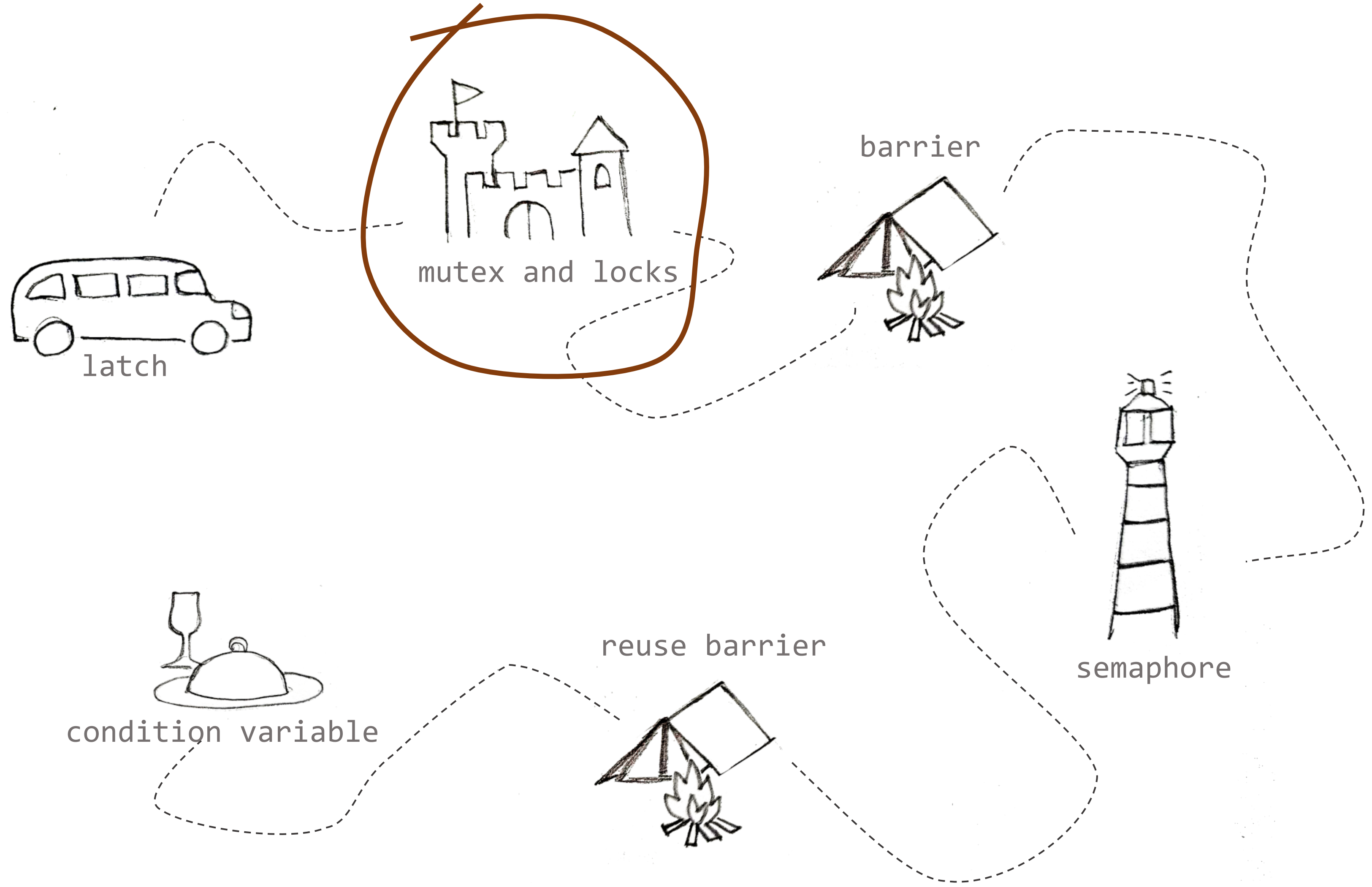


Everyone is seated thanks to LATCH

```
latch fullBus{bookedSeats};

array<jthread, hikersCnt> hikers;
ranges::for_each(hikers, [](auto &hiker) {
  hiker = jthread([]() { fullBus.arrive_and_wait(); });
});

jthread driver([]() {
  fullBus.wait();
  rideTheBus();
});

array<jthread, excusesCnt> excuses;
ranges::for_each(excuses, [](auto &excuse) {
  excuse = jthread([]() { fullBus.count_down(); });
});
```

- Single use meeting point is initialized with expected count of attendees
- Waiting thread will continue when the counter reach zero
- Decrement counter on arrival and wait for the rest
- you may wait for latch you didn't participate in
- You can decrement counter and continue without waiting for anyone

Get into medieval armor with LOCK



Get into medieval armor with LOCK

```
mutex armor;

//try the armor
armor.lock();
takeCoolArmoredSelfie();
if(busIsLeavingSoon)
{
  armor.unlock();
  return;
}
takeFewMoreSelfies();
armor.unlock();

shareSelfieOnline();
```

- Primitive providing exclusive access to critical section
- Only one thread can own mutex
- Others needs to wait until it will be freed before accessing protected critical section
- Code executed between `lock()` and `unlock()` is guaranteed to be accessed by only one thread at the same time
- Destructing locked mutex is UB
- Terminating thread which owns mutex is UB

Get into medieval armor with LOCK

```
mutex armor;

//try the armor
{
  lock_guard armorLock(armor);
  takeCoolArmoredSelfie();
}

shareSelfieOnline();
```

- Safe wrapper for mutex
- Will be locked during creation
- Unlocked while destructed
- After its definition whole scope will be locked

Get into medieval armor with LOCK

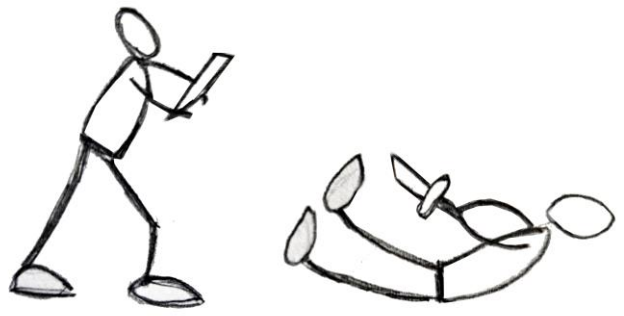
```
mutex armor;

//try the armor
{
  unique_lock armorLock(armor);
  takeCoolArmoredSelfie();

  if (wantBetterPictures) {
   armorLock.unlock();
   findGoodPhotographer();
   armorLock.lock();
   photoshooting();
  }
}
sharePicturesOnline();
```

- But sometimes you need to manage locking and unlocking by yourself
- With `unique_lock` you can lock and unlock as many times as you wish and still get the advantage of unlocking during destruction
- You can even define it as unlocked
- You can move lock to another function (unlike `lock_guard`)

Get into medieval armor with LOCK

```
shared_mutex armor;

//try the armor
if (wantToTryIt) {
  lock_guard armorLock(armor);
  takeCoolArmoredSelfie();
}
else {
  shared_lock armorLock(armor);
  takePictureOfArmorExhibition();
}
postPicturesOnline();
```

- Sometimes you know most of your threads will be just reading value you want to protect
- But there still is at least one which wants to write to it
- You want to allow all the readers to access this resource at the same time
- `shared_mutex` will allow you this scenario
- For writer you will feed simple `lock_guard` with it and it will get unique access
- For readers you feed `shared_lock` with that and access will be shared

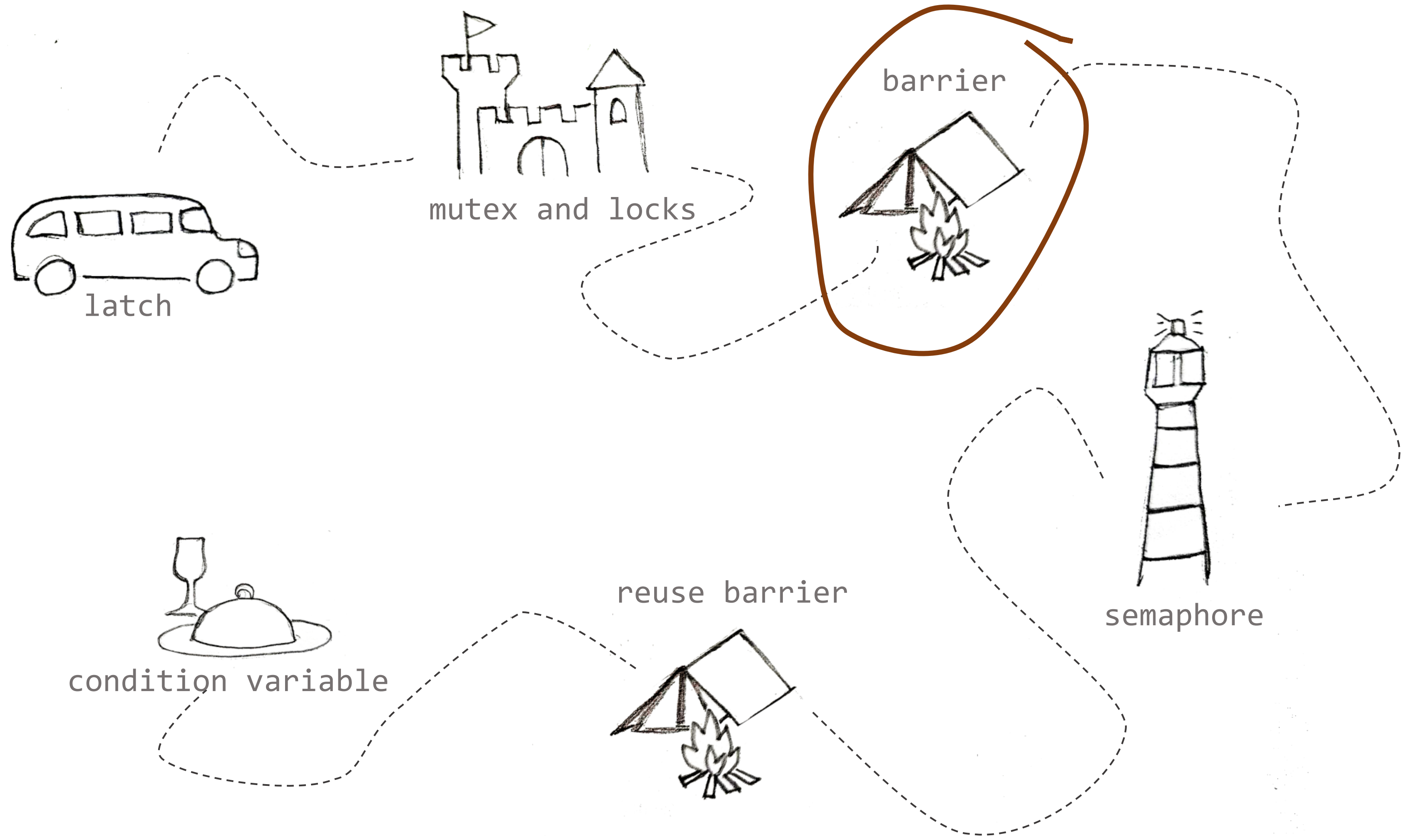


Always use the simplest possible version of locking!

Complexity comes with its price

In most cases you will be fine with `lock_guard`

Camp sleep over with BARRIER



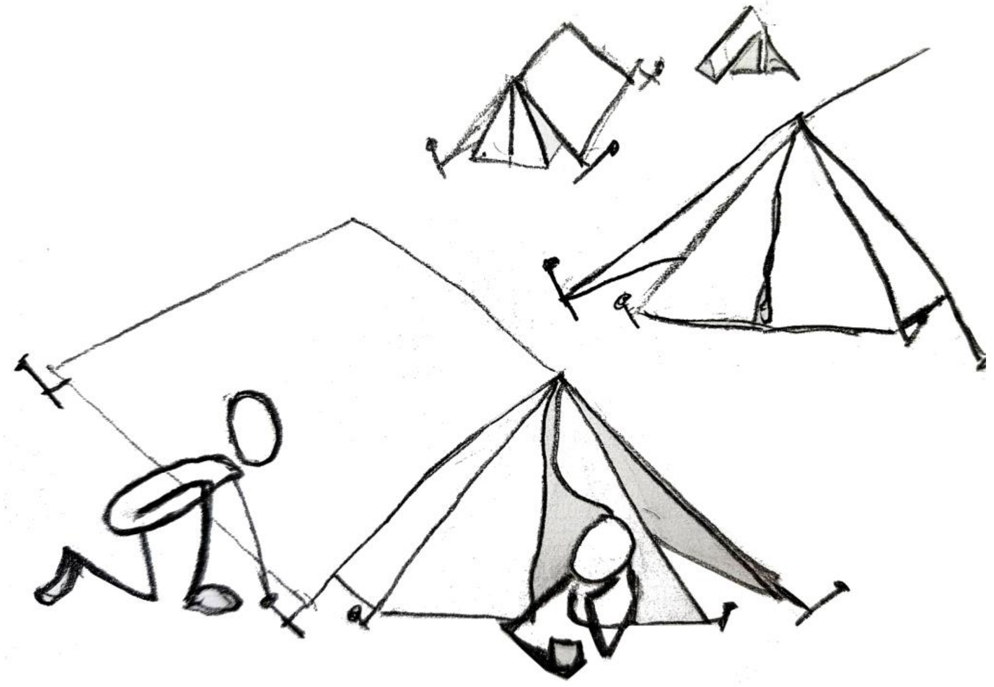
Camp sleep over with BARRIER

```
barrier expeditionCheck{hikersCnt, []()noexcept { planAnotherDay(); }};

//camp sleepover
if(feelSick){
  expeditionCheck.arrive_and_drop();
  return;
}
sleepWell();

expeditionCheck.arrive_and_wait();
```

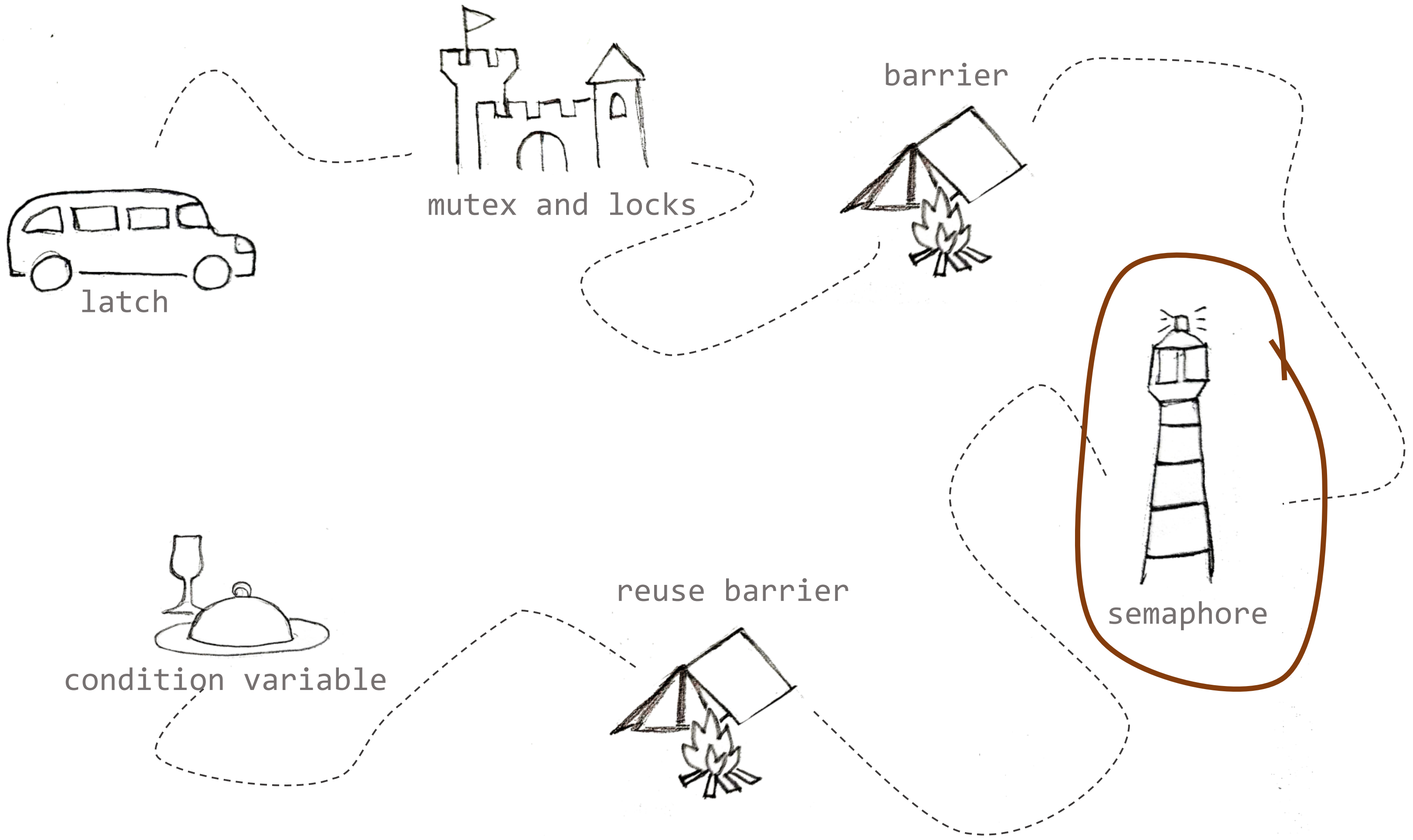
- Reusable meeting point is initialized with expected count of threads to arrive
- Competition function will be run on one of waiting threads once the counter reach 0
- Counter will be reset, all waiting threads will be unblocked and barrier may be reused again (next phase will start)
- Thread may arrive, but decide it doesn't need to wait
- It will also decrement expected count for next barrier phases
- Waiting thread will continue when the counter reach zero
- You can't wait for the barrier you didn't participate in

Camp sleep over with BARRIER

```
struct tentBarrier : public barrier<void(*)()noexcept> {
  tentBarrier() : barrier(tentPartsCnt, []()noexcept{ buildTent();}){}
};
array<tentBarrier, tentCnt> tents{};
barrier expeditionCheck{hikersCnt, []()noexcept { startAnotherDay(); }};

//camp sleepover
if(ranges::contains(sickHikersNames, hikerName)){
  tents[tentId].arrive_and_drop();
  expeditionCheck.arrive_and_drop();
  return;
}

auto token = tents[tentId].arrive();
eatSnack();
tents[tentId].wait(move(token));
sleepWell();

expeditionCheck.arrive_and_wait();
```



Visit lighthouse with SEMAPHORE



Visit lighthouse with SEMAPHORE

```
counting_semaphore lighthouseCapacity{ 6 };

//hungry guy
if (!lighthouseCapacity.try_acquire()) {
  eat();
  lighthouseCapacity.acquire();
  enjoyView();
  lighthouseCapacity.release();
  return;
}
quickVisit();
lighthouseCapacity.release();
eat();
```

- Primitive that allows up to defined count of threads to concurrently access critical section
- `binary_semaphore` is special case for maximal value of 1 and may be better optimized
- By acquiring you decrement counter of current visitors
- If the counter is 0 you will be blocked until some thread release its slot
- By releasing you increment internal counter and allow another thread to enter
- You may also try to acquire access just once (`try_acquire`) , try it for some time duration(`try_acquire_for`) or until given time(`try_acquire_until`)

Visit lighthouse with SEMAPHORE

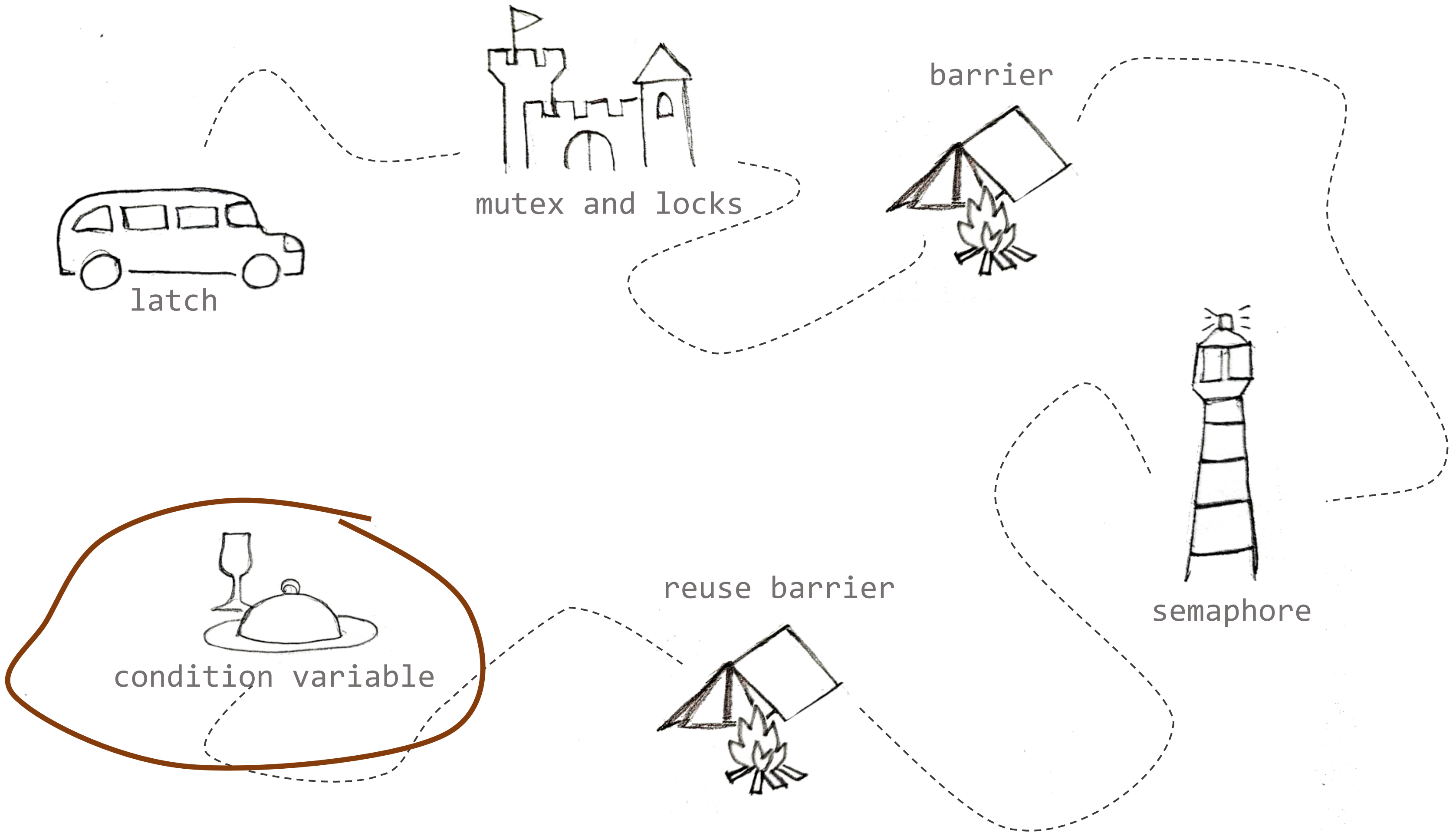
```
counting_semaphore lighthouseCapacity{ 6 };
latch completeFamily{familyVisit.size()};

//family ticket
if (headOfFamily)
  ranges::for_each(familyVisit, [](const auto& member){
    lighthouseSemaphore.acquire();});

completeFamily.arrive_and_wait();
enjoyView();
lighthouseSemaphore.release();
```

- There is no wrapper like `lock_guard` – you have to keep track of your acquires and releses
- `binary_semaphore` doesn't work the same way locks does
- With semaphore you can acquire access from one thread and release it by another

What will happen if we remove latch?

- `headOfFamily` may not come first and so others would be missing tickets and will violate rules by coming inside.
- Even if it won't cause any trouble inside lighthouse it will be problem when they will try to release permission they don't own
- They will accidentally increase the lighthouse capacity

Order lunch with CONDITION VARIABLE



Order lunch with CONDITION VARIABLE

```
mutex ordersMutex;
condition_variable orders;
int ordersToPlace = guestCnt;

jthread waiter{[]() {
  comeToTable();
  unique_lock ordersList(ordersMutex);
  orders.wait(ordersList, [] { return !ordersToPlace; });
  leaveTable();
}};

//guest
readMenu();
{
  lock_guard lk(ordersMutex);
  --ordersToPlace;
}
orders.notify_one();
```

- Synchronization primitive used together with mutex
- Waiting thread will be blocked until shared variable reach expected value
- Waiting thread will be notified about that change – no need to poll it
- For waiting you need to pass `unique_lock` or other mutex wrapper which can be locked and unlocked repeatedly
- Lock is acquired only during condition check than it is released to allow other threads to change shared variable
- Don’t use wait without check – you may get into problems during spurious wakeup
- Check is performed after notification arrives
- Value change may be done under `lock_guard`
- After you change value, you need to send notification to all or one of waiting threads

Summary on blocking synchronization primitives

Primitives defining happen-before relationship present in C++20

- Latch
- Mutexes and whole variety of locks
- Barrier
- Semaphore
- Condition variable

What they have in common

- Ensures execution safety by blocking further progress of threads
- Limits possible code reordering
- May introduce deadlock
- Require attention to correct releasing of acquired resources

Brief introduction into ATOMICS

- `atomic<T> x;`
- Allows you to change value atomically – without any risk of being interrupted by other thread
- May be of any trivially copyable, copy constructible and copy assignable type

- You can access them without utilizing previously mentioned primitives
- Often are mentioned as lock-free primitives, but they still may block internally (only `atomic_flag` is guaranteed to be lock-free)

- Only one operation is guaranteed to be performed without interruption
- Two subsequent atomic operation may be interleaved by other thread workload

```
atomic<int> x = 0;

if(x > maxCount)
  --x;            //x may have been changed by other thread since you evaluated condition
```

Operations on ATOMICS

For integer types you can use those basic operations

- fetch_add / operator++ / operator+=
- fetch_sub / operator-- / operator-=
- fetch_and / operator&=
- fetch_or / operator|=
- fetch_xor / operator^=

```
atomic<int> x = 0;

int oldX = x.fetch_add(3);
x--;
oldX = x.fetch_or(4);
x |= 1;
x += 1;
x = x + 1; //atomic load - add one to tmp – atomic store
```

For floating point type you can use `fetch_add` and `fetch_sub`

Please check with documentation which types exactly are supported. It changes with C++ versions

Operations on ATOMICS

```
atomic<int> hikersOnTop = 0;
atomic<int> postersLeft = hikersCnt;
thread_local int sandwichCnt = 5;


//hiker
GetOnTop();
hikersOnTop += 1;

ChoosePoster();
postersLeft = postersLeft - 1;

TakeBreak();
sandwichCnt = sandwichCnt – 1;
```

- `hikersOnTop` will execute fetch_add operation and behaves as expected

- `postersLeft` will execute sequence of operations

```
postersLeft.store(postersLeft.load() – 1);
```

- What may easily happen

```
postersCnt = postersLeft.load();        //5
//some hiker gets 3 posters postersLeft = 2
postersLeft.store(postersCnt - 1);      //4
```

- `sandwichCnt` isn't atomic there is no problem managing it this way

Operations on ATOMICS

```
atomic<int> hikersOnTop = 0;
atomic<int> postersLeft = hikersCnt;
thread_local int sandwichCnt = 5;

//hiker
GetOnTop();
hikersOnTop.fetch_add(1);

ChoosePoster();
postersLeft.fetch_sub(1);

TakeBreak();
sandwichCnt = sandwichCnt – 1;
```

- `hikersOnTop` will execute fetch_add operation and behaves as expected
- `postersLeft` will execute sequence of operations

```
postersLeft.store(postersLeft.load() – 1);
```

- What may easily happen

```
postersCnt = postersLeft.load();        //5
//some hiker gets 3 posters postersLeft = 2
postersLeft.store(postersCnt - 1);      //4
```

- `sandwichCnt` isn't atomic there is no problem managing it this way

Operations on ATOMICS

For atomic of all types you can perform those operations

- is_lock_free
- load / operator T
- store / operator=
- exchange

```
bool isLockFree = x.is_lock_free();
auto localX = x;    localX = x.load();
x = localX;         x.store(localX);
auto oldX = x.exchange(localX);
```

- compare_exchange_strong
- compare_exchange_weak

```
atomic<int> currentBet = 1;

//double the bet – if someone else doubles sooner give up
int betToBeDoubled = currentBet;
int myBet = betToBeDoubled << 1;

if(currentBet.compare_exchange_strong(betToBeDoubled, myBet))
  continueInGame();
else
  giveUp();
```

Operations on ATOMICS

For atomic of all types you can perform those operations

- is_lock_free
- load / operator T
- store / operator=
- exchange

```
bool isLockFree = x.is_lock_free();

auto localX = x;    localX = x.load();

x = localX;         x.store(localX);

auto oldX = x.exchange(localX);
```

- compare_exchange_strong
- **compare_exchange_weak**

```
atomic<int> currentBet = 1;

//double the bet no matter what
int betToBeDoubled = currentBet;
int myBet = betToBeDoubled << 1;

while(!currentBet.compare_exchange_weak(betToBeDoubled, myBet))
  myBet += (betToBeDoubled << 1) - myBet;
```

Operations on ATOMICS

```
atomic<bool> banana = true;

//hiker
bool expectsBanana = true;
if(banana.compare_exchange_strong(expectsBanana, false))
  eatBanana();

//villager
bool expectsBanana = true;
while(!banana.compare_exchange_weak(expectsBanana, false))
  expectsBanana = true;

//shop owner
bool expectsBanana = false;
if(!banana.compare_exchange_strong(expectsBanana, true))
  putNewBananasBackToBox();
```

- Hiker won’t come back repeatedly – he’s just passing by
- He is willing to wait in queue no matter the size – next shop is 20km away

- Villager may stop by every time he goes around
- He doesn’t want to wait large queue – prefers to try it again later when it will be less crowded

- Shop owner is adding missing goods time to time
- Depends on his nature if he waits strongly on weakly

Operations on ATOMICS

```
atomic<bool> banana = true;

//hiker
bool expectsBanana = true;
if(banana.exchange(false))
  eatBanana();

//villager
bool expectsBanana = true;
while(!banana.compare_exchange_weak(expectsBanana, false))
  expectsBanana = true;

//shop owner
bool expectsBanana = false;
if(!banana.compare_exchange_strong(expectsBanana, true))
  putNewBananasBackToBox();
```

- Hiker won’t come back repeatedly – he’s just passing by
- He is willing to wait in queue no matter the size – next shop is 20km away

- Villager may stop by every time he goes around
- He doesn’t want to wait large queue – prefers to try it again later when it will be less crowded

- Shop owner is adding missing goods time to time
- Depends on his nature if he waits strongly on weakly

Operations on ATOMICS

wait and notify_one / notify_all

- little bit different from what we know from condition_variables
- wait(old) blocks until the value of atomic is **different** than old
- wait is guaranteed to return only if value has changed – no spurious wake-ups
- notify_one / notify_all behaves same way as with condition_variables
- no need to pass any locks – we are in atomic world now

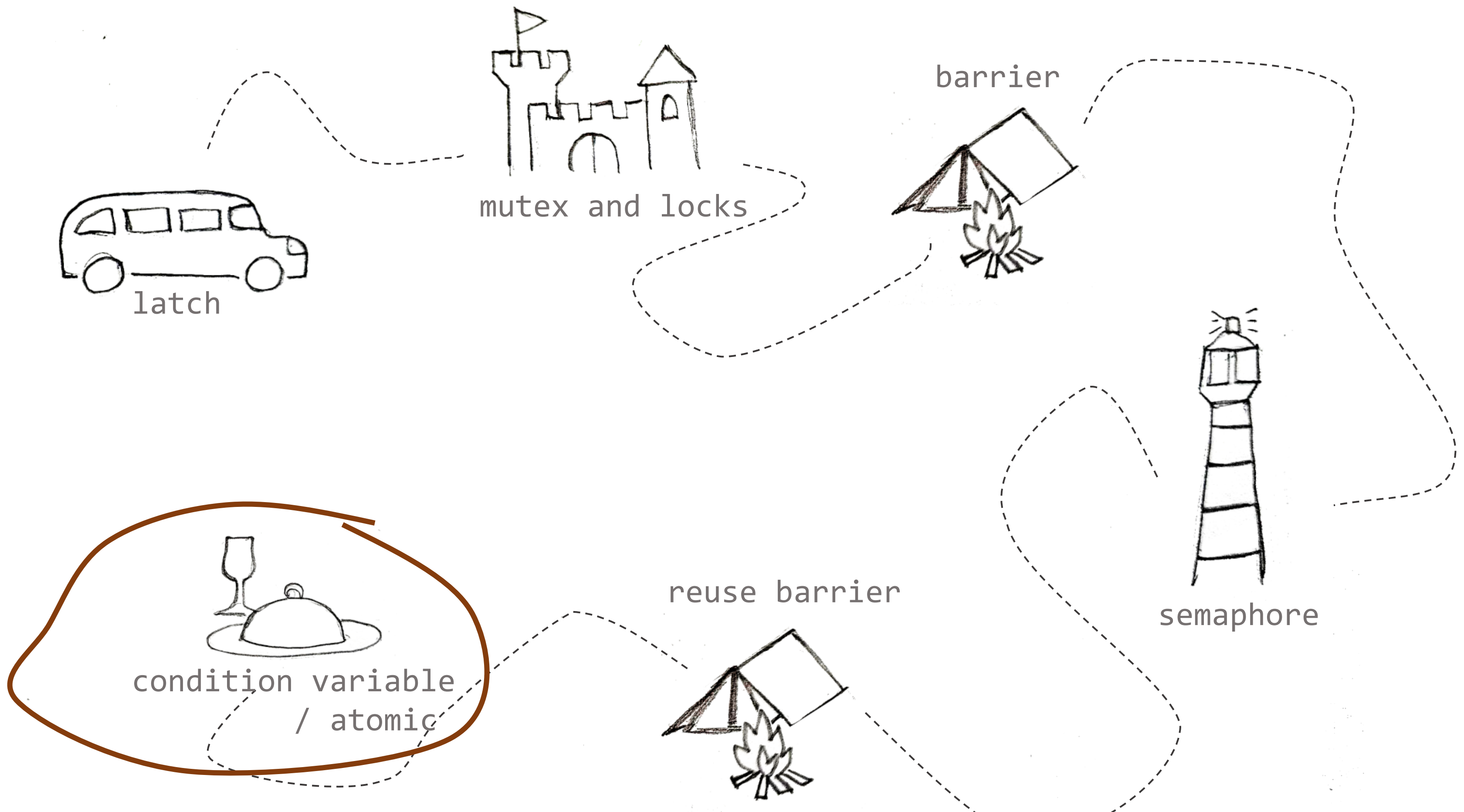
atomic_ref

- allows you to reference non-atomic variable as if it was atomic
- during lifetime of atomic_ref all operation on it are atomic
- lifetime of referenced object has to exceed lifetime of atomic_ref
- you are not allowed to access referenced object directly during atomic_ref lifetime

atomic<shared_ptr>

- new way allowing us atomically access shared pointers

Pay lunch with ATOMICS



Pay lunch with ATOMICS

```
atomic<int> billToPay = sumOfAllLunches;
atomic<bool> billCleaned = false;
thread_local int myLunchPrice = 0;

const int billBeforeMyPay = billToPay.fetch_sub(myLunchPrice);
if (billBeforeMyPay > myLunchPrice) {
  billCleaned.wait(false);
}
else{
  billCleaned = true;
  billCleaned.notify_all();
}
seeYouOnNextHike();
```

- `thread_local` means each thread has its own copy of that variable
- `fetch_sub` atomicaly subtract my price from bill and return old value – the one before subtraction
- Wait if there is still something to pay
- Waiting for value to be changed from the expected one – different logic than condition variables have!
- Notify everyone if I was the one who closed the bill

Call waiter with ATOMIC

```
atomic<bool> isWaiterPresent = false;
atomic<bool> allOrdersPlaced = false;
atomic<int> ordersToPlace = guestsCnt;

//guest
bool waiterExpectedState = false;
if(isWaiterPresent.compare_exchange_strong(waiterExpectedState, true)) {
  jthread waiter{[]() {
    comeToTable();
    allOrdersPlaced.wait(false);
    leaveTable();}};
}


readMenu();
if (--ordersToPlace == 0){
  allOrdersPlaced = true;
  allOrdersPlaced.notify_one();}
```

Replace condition variable with atomic

With atomic it’s the customer who is checking if there are some other orders to make

It’s because waiting check only change of value from expected and doesn’t evaluate custom condition

Call the waiter

Before he just somehow appeared and we didn’t investigate when he has spawned by the table

Now we can find out by `compare_exchange_strong` that we are first customer to order and so we have to call waiter to be able to start

Summary on ATOMICS

Atomics allow us to perform several basic operations on shared data without risk of data race

- Arithmetic and logical operations for interger and floating point data
- `load, store` and `exchange`, `compare_exchange_strong` and `compare_exchange_weak` for all atomic types
- wait and `notify_one / notify_all`

We may atomically manipulate with pointers using `atomic<shared_ptr>`

We can reference ordinary value by `atomic_ref` and operate with that as if it was atomic

We have overloaded operator for basic operations

Feels good while writing but are tricky to read - atomicity is not explicitly visible which may lead to unnecessary mistakes

Always keep in mind - two consecutive atomic operations doesn’t make atomic block

There are advanced options to explore on atomics

Before exploring those be sure you are really mastering what we’ve just covered

Summary

We found out how data flows through memory hierarchy

Why is our code reordered and what are the consequences for accessing shared memory

Why do we have memory model and what does it define

Which synchronizations primitives are available in C++20 to define happen-before relation

What are atomics and what are basic operations we can perform with them

We still haven't covered all possible topics on concurrency you still can explore

- Parallel algorithm
- Lock-free programming
- Memory-ordering
- Threads lifecycle

If you want to play with the complete hike example you can get it here

(but you have to run it locally)

Memory Model

Get your shared data under control

Jana Machutová