

$$MX = M \parallel (P_1 \parallel \dots \parallel P_n)$$

$$M = (\textit{lock} \rightarrow \textit{unlock} \rightarrow M)$$

$$P_i = (\textit{lock} \rightarrow \textit{begin_critical_region}_i \rightarrow \\ \textit{end_critical_region}_i \rightarrow \textit{unlock} \rightarrow \\ \textit{private_work}_i \rightarrow P_i)$$

- n Prozesse $P_1 \dots P_n$ wollen konkurrierend auf eine Ressource zugreifen, aber zu einem Zeitpunkt soll diese nur einem Prozess zur Verfügung stehen.
- Die Ressource wird von M verwaltet.
- Wegen dem \parallel -Operator wird unter den P_i jeweils ein Prozess nicht-deterministisch ausgewählt, der für \textit{lock} bereit ist.

task-queue.hpp

```
template<typename Task>
class TaskQueue {
public:
    void submit(Task task) {
        mutex.lock();
        queue.emplace_back(std::move(task));
        mutex.unlock();
    }
    bool fetch(Task& task) {
        mutex.lock();
        bool success;
        if (queue.empty()) {
            success = false;
        } else {
            success = true;
            task = std::move(queue.front());
            queue.pop_front();
        }
        mutex.unlock();
        return success;
    }
private:
    std::mutex mutex;
    std::deque<Task> queue;
};
```

task-queue.hpp

```
void submit(Task task) {  
    mutex.lock();  
    queue.emplace_back(std::move(task));  
    mutex.unlock();  
}
```

- Bei Threads wird ein gegenseitiger Ausschluss über sogenannte Mutex-Variablen (*mutual exclusion*) organisiert.
- Dies wird primär dazu genutzt, um auf gemeinsame Datenstrukturen (wie hier die *TaskQueue*) konkurrierend zuzugreifen.
- Sogenannte „kritische Regionen“ werden dann von *mutex.lock()* und *mutex.unlock* eingeschlossen.
- Das stellt sicher, dass sich nie mehr als ein Thread innerhalb einer kritischen Region befindet.

- Gemeinsame Datenstrukturen, auf die konkurrierend zugegriffen wird, sollten grundsätzlich in eine Klasse verpackt werden, bei der
 - ▶ die Daten alle **private** deklariert sind und
 - ▶ Zugriffe auf die Daten in den Methoden immer innerhalb einer kritischen Regionen stattfinden.
- Problem: Was passiert, wenn der Aufruf einer Funktion wie z.B. `queue.emplace_back(std::move(task))` zu einer Ausnahmenbehandlung führt? Dann wird die Ausführung von `submit` abrupt beendet, die `mutex` bleibt weiterhin gesperrt, so dass anschließend jeder Aufruf einer der Methoden dieser `TasksQueue` zu einem Deadlock führt.
- Die Lösung besteht in der Verwendung von RAII-Objekten.

```
template<typename Mutex>
class lock_guard {
public:
    lock_guard(Mutex& mutex) : mutex(mutex) {
        mutex.lock();
    }
    ~lock_guard() {
        mutex.unlock();
    }
    lock_guard(const lock_guard&) = delete;
    lock_guard& operator=(const lock_guard&) = delete;
private:
    Mutex& mutex;
};
```

- Der Standard bietet hierfür `std::lock_guard` an, das (etwas vereinfacht) so aussieht.

task-queue.hpp

```
template<typename Task>
class TaskQueue {
public:
    void submit(Task task) {
        std::lock_guard<std::mutex> lock(mutex);
        queue.emplace_back(std::move(task));
    }
    bool fetch(Task& task) {
        std::lock_guard<std::mutex> lock(mutex);
        if (queue.empty()) {
            return false;
        } else {
            task = std::move(queue.front());
            queue.pop_front();
            return true;
        }
    }
private:
    std::mutex mutex;
    std::deque<Task> queue;
};
```

philo.cpp

```
int main() {
    constexpr unsigned int PHILOSOPHERS = 5;
    std::thread philosopher[PHILOSOPHERS];
    std::mutex fork[PHILOSOPHERS];
    for (int i = 0; i < PHILOSOPHERS; ++i) {
        philosopher[i] = std::thread(Philosopher(i+1,
            fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS]));
    }
    for (int i = 0; i < PHILOSOPHERS; ++i) {
        philosopher[i].join();
    }
}
```

- Die Philosophen können auf Basis von Threads leicht umgesetzt werden, wobei die Gabeln als Mutex-Variablen repräsentiert werden.

philo.cpp

```
class Philosopher {
public:
    Philosopher(unsigned int id, std::mutex& left_fork,
               std::mutex& right_fork) :
        id(id), left_fork(left_fork), right_fork(right_fork) {
    }
    void operator()() { /* ... */ }
private:
    void print_status(const std::string& msg) const {
        std::lock_guard<std::mutex> lock(cout_mutex);
        std::cout << "philosopher [" << id << "]: " <<
            msg << std::endl;
    }
    unsigned int id;
    std::mutex& left_fork;
    std::mutex& right_fork;
};
```

- Dabei erhält jeder Philosoph Referenzen auf die ihm zugeordneten beiden Gabeln.


```
void operator()() {
    for (int i = 0; i < 5; ++i) {
        print_status("sits down at the table");
        {
            std::lock_guard<std::mutex> lock1(left_fork);
            print_status("picks up the left fork");
            {
                std::lock_guard<std::mutex> lock2(right_fork);
                print_status("picks up the right fork");
                {
                    print_status("is dining");
                }
            }
            print_status("returns the right fork");
        }
        print_status("returns the left fork");
        print_status("leaves the table");
    }
}
```

- Alle Philosophen nehmen hier zunächst die linke und dann die rechte Gabel. Bei dieser Vorgehensweise ist ein Deadlock möglich, wenn beispielsweise alle gleichzeitig die linke Gabel nehmen.

$$P = (P_1 \parallel P_2) \parallel MX_1 \parallel MX_2$$

$$MX_1 = (lock_1 \rightarrow unlock_1 \rightarrow MX_1)$$

$$MX_2 = (lock_2 \rightarrow unlock_2 \rightarrow MX_2)$$

$$P_1 = (lock_1 \rightarrow think_1 \rightarrow lock_2 \rightarrow unlock_2 \rightarrow unlock_1 \rightarrow P_1)$$

$$P_2 = (lock_2 \rightarrow think_2 \rightarrow lock_1 \rightarrow unlock_1 \rightarrow unlock_2 \rightarrow P_2)$$

- Wenn P_1 und P_2 beide sofort jeweils ihren ersten Lock erhalten, gibt es einen Deadlock:

$$\nexists t : t \in traces(P) \wedge t > \langle lock_1, think_1, lock_2, think_2 \rangle$$

- Eine der von Dijkstra vorgeschlagenen Deadlock-Vermeidungsstrategien (siehe EWD625) sieht die Definition einer totalen Ordnung aller MX_i vor, die z.B. durch die Indizes zum Ausdruck kommen kann.
- D.h. wenn MX_i und MX_j gehalten werden sollen, dann ist zuerst MX_i zu belegen, falls $i < j$, ansonsten MX_j .
- Dijkstra betrachtet den Ansatz selbst als unschön, weil damit eine willkürliche Anordnung erzwungen wird. Die Vorgehensweise ist nicht immer praktikabel, aber nicht selten eine sehr einfach umzusetzende Lösung.

```
struct Fork {
    unsigned id; std::mutex mutex;
};
/* ... */
int main() {
    constexpr unsigned int PHILOSOPHERS = 5;
    std::thread philosopher[PHILOSOPHERS];
    Fork fork[PHILOSOPHERS];
    for (int i = 0; i < PHILOSOPHERS; ++i) {
        fork[i].id = i;
    }
    for (int i = 0; i < PHILOSOPHERS; ++i) {
        philosopher[i] = std::thread(Philosopher(i+1,
            fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS]));
    }
    for (int i = 0; i < PHILOSOPHERS; ++i) {
        philosopher[i].join();
    }
}
```

- Um das umzusetzen, werden allen Ressourcen (das sind hier die Gabeln) eindeutige Nummern vergeben.

```
std::mutex* first; std::mutex* second;
std::string first_text; std::string second_text;
if (left_fork.id < right_fork.id) {
    first = &left_fork.mutex; second = &right_fork.mutex;
    first_text = "left"; second_text = "right";
} else {
    first = &right_fork.mutex; second = &left_fork.mutex;
    first_text = "right"; second_text = "left";
}
{
    std::lock_guard<std::mutex> lock1(*first);
    print_status("picks up the " + first_text + " fork");
    {
        std::lock_guard<std::mutex> lock2(*second);
        print_status("picks up the " + second_text + " fork");
        {
            print_status("is dining");
        }
    }
    print_status("returns the " + second_text + " fork");
}
print_status("returns the " + first_text + " fork");
print_status("leaves the table");
```

- C++ bietet zwei Klassen an, um Locks zu halten: `std::lock_guard` und `std::unique_lock`.
- Beide Varianten unterstützen die automatische Freigabe durch den jeweiligen Dekonstruktor.
- `std::lock_guard` ist eine reine RAII-Klasse: Die Ressource wird nur bei der Konstruktion belegt und die Freigabe erfolgt ausschließlich durch den Abbau.
- `std::unique_lock` bietet u.a. die Methoden `lock` und `unlock` an und einige spezielle Optionen bei der Konstruktion:

<code>std::defer_lock</code>	der Mutex wird noch nicht belegt
<code>std::try_to_lock</code>	es wird nicht-blockierend versucht, den Mutex zu belegen
<code>std::adopt_lock</code>	ein bereits belegter Mutex wird übernommen

- `std::shared_lock` kam mit C++14 und `std::shared_mutex` mit C++17 hinzu.

```
{
    std::unique_lock<std::mutex> lock1(left_fork, std::defer_lock);
    std::unique_lock<std::mutex> lock2(right_fork, std::defer_lock);
    std::lock(lock1, lock2);
    print_status("picks up both forks and is dining");
}
```

- C++ bietet mit `std::lock` eine Operation an, die beliebig viele Mutex-Variablen beliebigen Typs akzeptiert, und diese in einer vom System gewählten Reihenfolge belegt, die einen Deadlock vermeidet.
- Normalerweise erwartet `std::lock` Mutex-Variablen. `std::unique_lock` ist eine Verpackung, die wie eine Mutex-Variable verwendet werden kann.
- Zunächst nehmen die beiden `std::unique_lock`-Objekte die Mutex-Variablen jeweils in Beschlag, ohne eine `lock`-Operation auszuführen (`std::defer_lock`). Danach werden nicht die originalen Mutex-Variablen, sondern die `std::unique_lock`-Objekte an `std::lock` übergeben.
- Diese Variante ist umfassend auch gegen Ausnahmenbehandlungen abgesichert.

- Ein Monitor ist eine Klasse, bei der maximal ein Thread eine Methode aufrufen kann.
- Wenn weitere Threads konkurrierend versuchen, eine Methode aufzurufen, werden sie solange blockiert, bis sie alleinigen Zugriff haben (gegenseitiger Ausschluss).
- Der Begriff und die zugehörige Idee gehen auf einen Artikel von 1974 von C. A. R. Hoare zurück.
- Aber manchmal ist es sinnvoll, den Aufruf einer Methode von einer weiteren Bedingung abhängig zu machen,

- Bei Monitoren können Methoden auch mit Bedingungen versehen werden, d.h. eine Methode kommt nur dann zur Ausführung, wenn die Bedingung erfüllt ist.
- Wenn die Bedingung nicht gegeben ist, wird die Ausführung der Methode solange blockiert, bis sie erfüllt ist.
- Eine Bedingung sollte nur von dem internen Zustand eines Objekts abhängen.
- Bedingungsvariablen sind daher private Objekte eines Monitors mit den Methoden *wait*, *notify_one* und *notify_all*.
- Bei *wait* wird der aufrufende Thread solange blockiert, bis ein anderer Thread bei einer Methode des Monitors *notify_one* oder *notify_all* aufruft. (Bei *notify_all* können alle, die darauf gewartet haben, weitermachen, bei *notify_one* nur ein Thread.)
- Eine Notifizierung ohne darauf wartende Threads ist wirkungslos.

```

class Monitor {
public:
    void some_method() {
        std::unique_lock<std::mutex> lock(mutex);
        while (! /* some condition */) {
            condition.wait(lock);
        }
        // ...
    }
    void other_method() {
        std::unique_lock<std::mutex> lock(mutex);
        // ...
        condition.notify_one();
    }
private:
    std::mutex mutex;
    std::condition_variable condition;
};
    
```

- Bei der C++11-Standardbibliothek ist eine Bedingungsvariable immer mit einer Mutex-Variablen verbunden.
- *wait* gibt den Lock frei, wartet auf die Notifizierung, wartet dann erneut auf einen exklusiven Zugang und kehrt dann zurück.

Verknüpfung von Bedingungs- und Mutex-Variablen 114

- Die Methoden *notify_one* oder *notify_all* sind wirkungslos, wenn kein Thread auf die entsprechende Bedingung wartet.
- Wenn ein Thread feststellt, dass gewartet werden muss und danach wartet, dann gibt es ein Fenster zwischen der Feststellung und dem Aufruf von *wait*.
- Wenn innerhalb des Fensters *notify_one* oder *notify_all* aufgerufen wird, bleiben diese Aufrufe wirkungslos und beim anschließenden *wait* kann es zu einem Deadlock kommen, da dies auf eine Notifizierung wartet, die nun nicht mehr kommt.
- Damit das Fenster völlig geschlossen wird, muss *wait* als atomare Operation zuerst den Thread in die Warteschlange einreihen und erst dann den Lock freigeben.
- Bei *std::condition_variable* muss der Lock des Typs *std::unique_lock<std::mutex>* sein. Für andere Locks gibt es die u.U. weniger effiziente Alternative *std::condition_variable_any*.

$$M \parallel (P_1 \parallel P_2 \parallel CL) \parallel C$$

$$M = (\text{lock} \rightarrow \text{unlock} \rightarrow M)$$

$$P_1 = (\text{lock} \rightarrow \text{wait} \rightarrow \text{resume} \rightarrow \\ \text{critical_region}_1 \rightarrow \text{unlock} \rightarrow P_1)$$

$$P_2 = (\text{lock} \rightarrow \text{critical_region}_2 \rightarrow \\ (\text{notify} \rightarrow \text{unlock} \rightarrow P_2 \mid \text{unlock} \rightarrow P_2))$$

$$CL = (\text{unlock}_C \rightarrow \text{unlock} \rightarrow \text{unlocked}_C \rightarrow \\ \text{lock}_C \rightarrow \text{lock} \rightarrow \text{locked}_C \rightarrow CL)$$

$$C = (\text{wait} \rightarrow \text{unlock}_C \rightarrow \text{unlocked}_C \rightarrow \text{notify} \rightarrow \\ \text{lock}_C \rightarrow \text{locked}_C \rightarrow \text{resume} \rightarrow C)$$

- Einfacher Fall mit M für die Mutex-Variablen, einem Prozess P_1 , der auf eine Bedingungsvariable wartet, einem Prozess P_2 , der notifiziert oder es auch sein lässt, und der Bedingungsvariablen C , die hilfsweise CL benötigt, um gemeinsam mit P_1 und P_2 um die Mutexvariable konkurrieren zu können.

- Wenn notwendig, können auch eigene Klassen für Locks definiert werden.
- Die Template-Klasse `std::lock_guard` akzeptiert eine beliebige Lock-Klasse, die mindestens folgende Methoden unterstützt:

void `lock()` blockiere, bis der Lock reserviert ist
void `unlock()` gib den Lock wieder frei

- Typhierarchien und virtuelle Methoden werden hierfür nicht benötigt, da hier statischer Polymorphismus vorliegt, bei dem mit Hilfe von Templates alles zur Übersetzzeit erzeugt und festgelegt wird.
- In einigen Fällen (wie etwa die Übergabe an `std::lock`) wird auch noch folgende Methode benötigt:
bool `try_lock()` versuche, nicht-blockierend den Lock zu reservieren

resource-lock.hpp

```
class ResourceLock {
public:
    ResourceLock(unsigned int capacity) : capacity(capacity), used(0) {
    }
    void lock() {
        std::unique_lock<std::mutex> lock(mutex);
        while (used == capacity) {
            released.wait(lock);
        }
        ++used;
    }
    void unlock() {
        std::unique_lock<std::mutex> lock(mutex);
        assert(used > 0);
        --used;
        released.notify_one();
    }
private:
    const unsigned int capacity;
    unsigned int used;
    std::mutex mutex;
    std::condition_variable released;
};
```

philo4.cpp

```
constexpr unsigned int PHILOSOPHERS = 5;
ResourceLock rlock(PHILOSOPHERS-1);
std::thread philosopher[PHILOSOPHERS];
std::mutex fork[PHILOSOPHERS];
for (int i = 0; i < PHILOSOPHERS; ++i) {
    philosopher[i] = std::thread(Philosopher(i+1,
        fork[i], fork[(i + PHILOSOPHERS - 1) % PHILOSOPHERS], rlock));
}
```

- Mit Hilfe eines *ResourceLock* lässt sich das Philosophenproblem mit Hilfe eines Dieners lösen.
- Bei n Philosophen lassen die Diener zu, dass sich $n - 1$ Philosophen hinsetzen.

philo4.cpp

```
void operator()() {
    for (int i = 0; i < 5; ++i) {
        print_status("comes to the table");
        {
            std::lock_guard<ResourceLock> lock(rlock);
            print_status("got permission to sit down at the table");
            {
                std::lock_guard<std::mutex> lock1(left_fork);
                print_status("picks up the left fork");
                {
                    std::lock_guard<std::mutex> lock2(right_fork);
                    print_status("picks up the right fork");
                    {
                        print_status("is dining");
                    }
                }
                print_status("returns the right fork");
            }
            print_status("returns the left fork");
        }
        print_status("leaves the table");
    }
}
```