

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра Програмної інженерії

Звіт
з лабораторної роботи №8
з дисципліни: «Операційні системи»
з теми: «Керування потоками одного та декількох процесів»

Виконали:

ст. гр. ПЗПІ-23-2

Ситник Є. С.

Малишкін. А. С.

Перевірила:

доц. каф. ПІ,

Мельнікова Р. В.,

8 КЕРУВАННЯ ПОТОКАМИ ОДНОГО ТА ДЕКІЛЬКОХ ПРОЦЕСІВ

8.1 Мета роботи

Метою даної лабораторної роботи є вивчення створення процесів та потоків при виконанні програм.

8.2 Хід роботи

Оскільки ми використовуємо операційну систему Linux, дану лабораторну роботу нами було виконано із використанням крос-платформних функцій стандартної бібліотеки C++, за виключенням останнього завдання, для якого було задіяно засоби POSIX API, адже стандартна бібліотека не надає засобів синхронізації процесів.

8.2.1 Задача №1

Мета даного завдання – розробити програму, в якій декілька «виробників» (потоків) будуть знаходити імена файлів і додавати їх до спільної «черги». Одночасно декілька «споживачів» (інших потоків) братимуть ці імена файлів з черги і виводитимуть вміст кожного файлу.

Для перевірки роботи програми використаємо 2 потоки виробника та 4 потоки споживача, які будуть обробляти 3 файли.

```
int main() {
    std::string prefix = "./task-1/files";

    th_safe::queue<std::string> queue;

    std::vector<std::string> file_list = {
        prefix + "/file1.txt",
        prefix + "/file2.txt",
        prefix + "/file3.txt"
    };

    for (auto &&name : file_list) {
        std::ofstream(name) << std::format("Hello from {}\n", name);
    }

    std::vector<std::jthread> producers;
    for (std::int32_t i : std::views::iota(0, 2)) {
        producers.emplace_back(
            workers::producer,
            std::ref(queue),
            std::ref(file_list),
```

```

        i
    );
}

std::vector<std::jthread> consumers;
for (std::int32_t i : std::views::iota(0, 4)) {
    consumers.emplace_back(
        workers::consumer,
        std::ref(queue),
        i
    );
}

producers.clear();
queue.done();
}

```

Для зручнішої роботи із чергою з стандартної бібліотеки в багато-поточковому контексті створимо клас обгортку, що реалізує методи взаємодії із чергою з використанням м'ютексів.

```

namespace th_safe {
template <typename T> class queue {
public:
    void push(const T &value) {
        {
            std::lock_guard lock(mutex_);
            queue_.push(value);
        }
        cond_var_.notify_one();
    }

    std::optional<T> pop() {
        std::unique_lock lock(mutex_);
        cond_var_.wait(
            lock,
            [this] { return !queue_.empty() || done_; }
        );

        if (queue_.empty())
            return std::nullopt;

        T value = queue_.front();
        queue_.pop();
        return value;
    }

    void done() {
        {
            std::lock_guard lock(mutex_);
            done_ = true;
        }
        cond_var_.notify_all();
    }
}

```

```
private:
    std::queue<T> queue_;
    std::mutex mutex_;
    std::condition_variable cond_var_;
    bool done_ = false;
};
} // namespace th_safe
```

Виробники додають файли в чергу, а споживачі зчитують їхній вміст.

```
namespace workers {
void producer(
    th_safe::queue<std::string> &q,
    const std::vector<std::string> &files,
    int id
) {
    for (const auto &file : files) {
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        q.push(file);
        std::print("Produced ({}): {}\n", id, file);
    }
}

void consumer(th_safe::queue<std::string> &q, int id) {
    while (true) {
        auto item = q.pop();
        if (!item.has_value())
            break;

        std::ifstream file(item.value());
        if (!file) {
            std::print("Failed to open: {}\n", item.value());
            continue;
        }

        std::print("Consumed ({}): {}\n", id, item.value());
        for (std::string line; std::getline(file, line);) {
            std::print("\t{}\n", line);
        }
    }
}
} // namespace workers
```

Перевіримо правильність роботи програми

```
~/D/n/s/0/1/src >>> make task-1
Produced (1): ./task-1/files/file1.txt
Produced (0): ./task-1/files/file1.txt
Consumed (1): ./task-1/files/file1.txt
    Hello from ./task-1/files/file1.txt
Consumed (0): ./task-1/files/file1.txt
    Hello from ./task-1/files/file1.txt
Produced (1): ./task-1/files/file2.txt
Produced (0): ./task-1/files/file2.txt
Consumed (3): ./task-1/files/file2.txt
    Hello from ./task-1/files/file2.txt
Consumed (2): ./task-1/files/file2.txt
    Hello from ./task-1/files/file2.txt
Produced (1): ./task-1/files/file3.txt
Produced (0): ./task-1/files/file3.txt
Consumed (1): ./task-1/files/file3.txt
    Hello from ./task-1/files/file3.txt
Consumed (0): ./task-1/files/file3.txt
    Hello from ./task-1/files/file3.txt
```

Рисунок 8.1 – Результат виконання програми

8.2.2 Задача №2

Мета даного завдання – створити програму з потоками «читачів» та «письменників». Потоки-письменники будуть додавати нові записи в кінець спільного списку новин. Паралельно потоки-читачі будуть отримувати та читати останню додану новину з цього ж списку.

Для перевірки роботи програми використаємо 1 потік письменник та 3 потоки читачі.

```
int main() {
    th_safe::vector<std::string> news;
    std::atomic_bool done = false;

    std::jthread writer_thread([&] {
        workers::writer(news, 10);
        done = true;
    });
```

```

std::vector<std::jthread> readers;
for (std::int32_t id : std::views::iota(1, 4)) {
    readers.emplace_back(workers::reader, std::cref(news),
std::ref(done), id);
}
}

```

Для зручнішої роботи із вектором з стандартної бібліотеки в багато-поточковому контексті створимо клас обгортку, що реалізує методи взаємодії із вектором з використанням м'ютексів.

```

namespace th_safe {
    template<typename T>
    class vector {
    public:
        void push_back(const T& value) {
            std::unique_lock lock(mutex_);
            data_.push_back(value);
        }

        std::optional<T> back() const {
            std::shared_lock lock(mutex_);
            if (data_.empty()) return std::nullopt;
            return data_.back();
        }

        std::size_t size() const {
            std::shared_lock lock(mutex_);
            return data_.size();
        }

    private:
        mutable std::shared_mutex mutex_;
        std::vector<T> data_;
    };
}

```

Письменник додає новини в вектор, а читачі зчитують з вектора останню додану новину.

```

namespace workers {
    void writer(
        th_safe::vector<std::string> &news_vector,
        std::int32_t count = 10
    ) {
        for (std::int32_t i : std::views::iota(0, count)) {
            std::this_thread::sleep_for(std::chrono::milliseconds(200));

            auto news = std::format("News item {}", i + 1);
            news_vector.push_back(news);
            std::print("Writer: added '{}'\n", news);
        }
    }
}

```

```

}

void reader(
    const th_safe::vector<std::string> &news_vector,
    std::atomic_bool &done,
    std::int32_t id
) {
    std::size_t prev_size = 0;

    while (true) {
        std::this_thread::sleep_for(std::chrono::milliseconds(200));
        auto last = news_vector.back();

        if (!last.has_value()) {
            std::print("Reader {}: no news yet\n", id);
            continue;
        }

        std::size_t current_size = news_vector.size();
        if (current_size == prev_size && !done)
            continue;
        if (current_size == prev_size && done)
            break;

        std::print("Reader {}: reads '{}'\n", id, *last);
        prev_size = current_size;
    }

    std::print("Reader {}: finished.\n", id);
}
} // namespace workers

```

Перевіримо правильність роботи програми

```
~/D/n/s/0/1/src >> make task-2
Reader 2: no news yet
Reader 3: reads 'News item 1'
Writer: added 'News item 1'
Reader 1: no news yet
Reader 2: reads 'News item 1'
Writer: added 'News item 2'
Reader 1: reads 'News item 2'
Reader 3: reads 'News item 2'
Reader 1: reads 'News item 3'
Writer: added 'News item 3'
Reader 2: reads 'News item 2'
Reader 3: reads 'News item 3'
Writer: added 'News item 4'
Reader 2: reads 'News item 4'
Reader 1: reads 'News item 4'
Reader 3: reads 'News item 4'
Writer: added 'News item 5'
Reader 1: reads 'News item 5'
Writer: added 'News item 6'
Reader 3: reads 'News item 5'
Reader 2: reads 'News item 5'
Reader 1: reads 'News item 6'
Writer: added 'News item 7'
Reader 3: reads 'News item 7'
Reader 2: reads 'News item 7'
Reader 1: reads 'News item 7'
Writer: added 'News item 8'
Reader 3: reads 'News item 8'
Reader 2: reads 'News item 8'
Reader 1: reads 'News item 8'
Reader 2: reads 'News item 9'
Reader 3: reads 'News item 9'
Writer: added 'News item 9'
Reader 1: reads 'News item 9'
Writer: added 'News item 10'
Reader 2: reads 'News item 10'
Reader 1: reads 'News item 10'
Reader 3: reads 'News item 10'
Reader 1: finished.
Reader 2: finished.
Reader 3: finished.
```

Рисунок 8.2 – Результат виконання програми

8.2.3 Задача №3

Мета даного завдання – реалізувати програму, яка моделює відому проблему «обідаючих філософів» за допомогою потоків. Кожен потік представлятиме філософа, який проходить повний цикл дій: думає, бере по черзі дві виделки, обідає, а потім кладе виделки на місце. Цей цикл має повторюватися задану кількість разів для кожного філософа.

Для перевірки роботи програми використаємо 5 потоків філософів та 3 повторення.

```
int main() {
    const std::size_t num_philosophers = 5;
    const std::size_t rounds = 3;

    th_safe::forks table(num_philosophers);

    std::vector<std::jthread> philosophers;
    for (auto id : std::views::iota(0UL, num_philosophers)) {
        philosophers.emplace_back(
            workers::philosopher,
            id,
            std::ref(table),
            rounds
        );
    }
}
```

Для зручної роботи із «столом» в багато-потоківому контексті створимо клас, що реалізує методи взаємодії із столом з використанням м'ютексів.

```
namespace th_safe {
class forks {
public:
    explicit forks(std::size_t count) : mutexes_(count) {}

    std::mutex &left(std::size_t i) { return mutexes_[i]; }

    std::mutex &right(std::size_t i) {
        return mutexes_[(i + 1) % mutexes_.size()];
    }

    std::size_t size() const { return mutexes_.size(); }

private:
    std::vector<std::mutex> mutexes_;
};
} // namespace th_safe
```

Кожен з філософів по черзі із певною затримкою виконує одну із можливих дій.

```
namespace workers {
void philosopher(
    std::size_t id,
    th_safe::forks &table,
    std::size_t iterations = 3
) {
    std::mt19937 rng(id + std::random_device{}());
    std::uniform_int_distribution<> think_time(100, 300);
    std::uniform_int_distribution<> eat_time(100, 200);

    for (std::size_t round = 0; round < iterations; ++round) {
        std::print(
            "Philosopher {} is thinking (round {})\n",
            id,
            round + 1
        );
        std::this_thread::sleep_for(
            std::chrono::milliseconds(think_time(rng))
        );

        std::mutex &left_fork = table.left(id);
        std::mutex &right_fork = table.right(id);

        if (std::addressof(left_fork) < std::addressof(right_fork)) {
            std::scoped_lock lock(left_fork, right_fork);
            std::print(
                "Philosopher {} is eating (round {})\n",
                id,
                round + 1
            );

            std::this_thread::sleep_for(
                std::chrono::milliseconds(eat_time(rng))
            );
            std::print(
                "Philosopher {} finished eating (round {})\n",
                id,
                round + 1
            );
        } else {
            std::scoped_lock lock(right_fork, left_fork);
            std::print("Philosopher {
                } is eating (round {})\n", id, round + 1);

            std::this_thread::sleep_for(
                std::chrono::milliseconds(eat_time(rng))
            );
            std::print(
                "Philosopher {} finished eating (round {})\n",
                id,
                round + 1
            );
        }
    }
}
```

```
std::print("Philosopher {} leaves the table.\n", id);
}
} // namespace workers
```

Перевіримо правильність роботи програми

```
~/D/n/s/0/1/src >>> make task-3
Philosopher 0 is thinking (round 1)
Philosopher 1 is thinking (round 1)
Philosopher 3 is thinking (round 1)
Philosopher 2 is thinking (round 1)
Philosopher 0 is eating (round 1)
Philosopher 2 is eating (round 1)
Philosopher 0 finished eating (round 1)
Philosopher 0 is thinking (round 2)
Philosopher 2 finished eating (round 1)
Philosopher 2 is thinking (round 2)
Philosopher 3 is eating (round 1)
Philosopher 1 is eating (round 1)
Philosopher 3 finished eating (round 1)
Philosopher 3 is thinking (round 2)
Philosopher 1 finished eating (round 1)
Philosopher 1 is thinking (round 2)
Philosopher 0 is eating (round 2)
Philosopher 2 is eating (round 2)
Philosopher 0 finished eating (round 2)
Philosopher 0 is thinking (round 3)
Philosopher 2 finished eating (round 2)
Philosopher 2 is thinking (round 3)
Philosopher 3 is eating (round 2)
Philosopher 1 is eating (round 2)
Philosopher 1 finished eating (round 2)
Philosopher 1 is thinking (round 3)
Philosopher 3 finished eating (round 2)
Philosopher 3 is thinking (round 3)
Philosopher 0 is eating (round 3)
Philosopher 2 is eating (round 3)
Philosopher 0 finished eating (round 3)
Philosopher 0 leaves the table.
Philosopher 2 finished eating (round 3)
Philosopher 2 leaves the table.
Philosopher 1 is eating (round 3)
Philosopher 3 is eating (round 3)
Philosopher 1 finished eating (round 3)
Philosopher 1 leaves the table.
Philosopher 3 finished eating (round 3)
Philosopher 3 leaves the table.
```

Рисунок 8.3 – Результат виконання програми

8.2.4 Завдання на найвищу оцінку

Мета даного завдання – створити окрему програму, яка буде багаторазово запускати одну з попередніх програм. Ці запуски повинні відбуватися за певним заданим розкладом, використовуючи механізм, такий як `WaitableTimer`.

Виконаємо завдання для програми №1.

Перевіримо роботу із 3 окремим процесами.

```
int main() {
    constexpr std::int32_t runs = 3;
    constexpr auto interval = std::chrono::seconds();

    for (std::int32_t i : std::views::iota(0, runs)) {
        std::print("\nLaunching: ./launcher/build/app (run {})\n", i +
1);

        std::int32_t result = std::system("./launcher/build/app");
        if (result != 0) {
            std::print("Run {} failed with code {}\n", i + 1, result);
        }

        std::this_thread::sleep_for(interval);
    }

    return 0;
}
```

Щоб кілька окремих процесів мали доступ до однієї черги використаємо функцію «`mmap`». Для синхронізації кількох процесів скористаємося семафорами.

```
namespace ipc {

constexpr size_t MAX_PATH_LEN = 256;
constexpr size_t QUEUE_CAPACITY = 128;

struct shared_queue {
    sem_t mutex;
    sem_t slots;
    sem_t items;
    size_t head;
    size_t tail;
    char data[QUEUE_CAPACITY][MAX_PATH_LEN];
    bool initialized;
};

class queue {
public:
    queue(const char *shm_name) {
        shm_fd_ = ::shm_open(shm_name, O_CREAT | O_RDWR, 0666);
        if (shm_fd_ < 0)
            throw std::runtime_error("shm_open failed");
    }
};
```

```

    if (::ftruncate(shm_fd_, sizeof(shared_queue)) < 0)
        throw std::runtime_error("ftruncate failed");

    ptr_ = static_cast<shared_queue *>(
        ::mmap(
            nullptr,
            sizeof(shared_queue),
            PROT_READ | PROT_WRITE,
            MAP_SHARED,
            shm_fd_,
            0
        )
    );
    if (ptr_ == MAP_FAILED)
        throw std::runtime_error("mmap failed");

    if (!ptr_>initialized) {
        initialize();
        ptr_>initialized = true;
    }
}

~queue() {
    ::munmap(ptr_, sizeof(shared_queue));
    ::close(shm_fd_);
}

void push(const std::string &s) {
    if (s.size() >= MAX_PATH_LEN)
        throw std::length_error("path too long");
    ::sem_wait(&ptr_>slots);
    ::sem_wait(&ptr_>mutex);

    std::strncpy(ptr_>data[ptr_>tail], s.c_str(), MAX_PATH_LEN);
    ptr_>tail = (ptr_>tail + 1) % QUEUE_CAPACITY;

    ::sem_post(&ptr_>mutex);
    ::sem_post(&ptr_>items);
}

bool pop(std::string &out) {
    ::sem_wait(&ptr_>items);
    ::sem_wait(&ptr_>mutex);

    char buf[MAX_PATH_LEN];
    std::strncpy(buf, ptr_>data[ptr_>head], MAX_PATH_LEN);
    ptr_>head = (ptr_>head + 1) % QUEUE_CAPACITY;

    ::sem_post(&ptr_>mutex);
    ::sem_post(&ptr_>slots);

    out = buf;
    return true;
}

private:
    int shm_fd_;

```

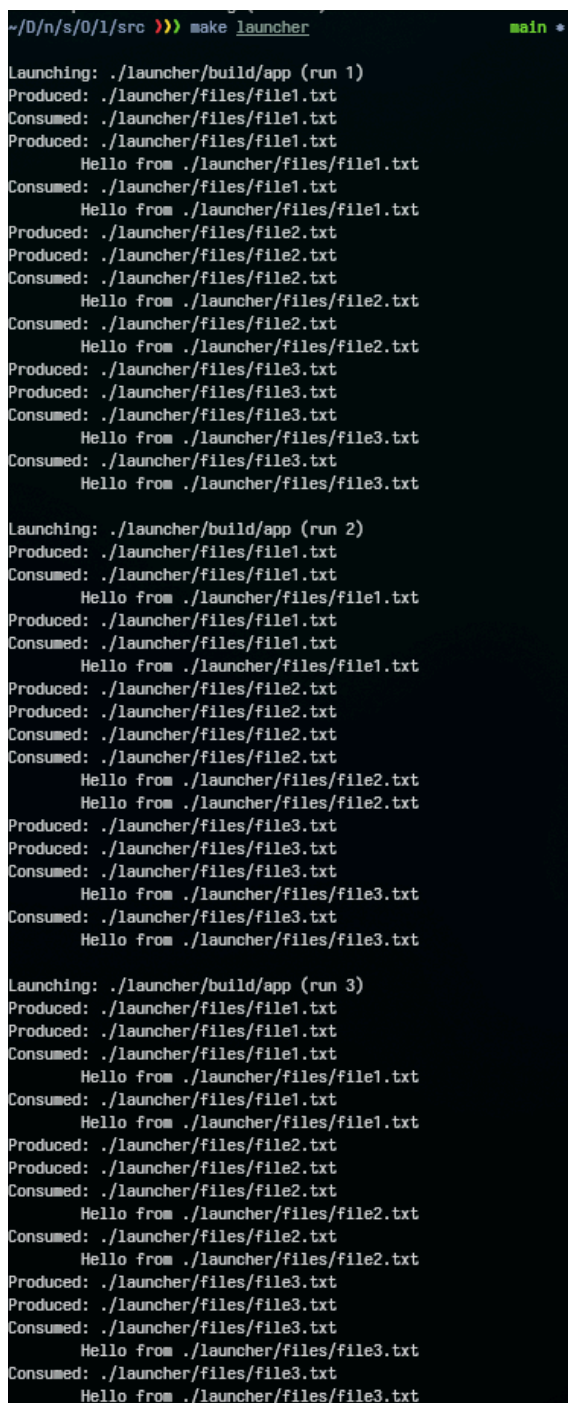
```

shared_queue *ptr_;

void initialize() {
    ::sem_init(&ptr_>mutex, 1, 1);
    ::sem_init(&ptr_>slots, 1, QUEUE_CAPACITY);
    ::sem_init(&ptr_>items, 1, 0);
    ptr_>head = ptr_>tail = 0;
}
};
} // namespace ipc

```

Перевіримо роботу зміненої програми.



```

~/D/n/s/0/1/src >> make launcher
Launching: ./launcher/build/app (run 1)
Produced: ./launcher/files/file1.txt
Consumed: ./launcher/files/file1.txt
Produced: ./launcher/files/file1.txt
Hello from ./launcher/files/file1.txt
Consumed: ./launcher/files/file1.txt
Hello from ./launcher/files/file1.txt
Produced: ./launcher/files/file2.txt
Produced: ./launcher/files/file2.txt
Consumed: ./launcher/files/file2.txt
Hello from ./launcher/files/file2.txt
Consumed: ./launcher/files/file2.txt
Hello from ./launcher/files/file2.txt
Produced: ./launcher/files/file3.txt
Produced: ./launcher/files/file3.txt
Consumed: ./launcher/files/file3.txt
Hello from ./launcher/files/file3.txt
Consumed: ./launcher/files/file3.txt
Hello from ./launcher/files/file3.txt

Launching: ./launcher/build/app (run 2)
Produced: ./launcher/files/file1.txt
Consumed: ./launcher/files/file1.txt
Hello from ./launcher/files/file1.txt
Produced: ./launcher/files/file1.txt
Consumed: ./launcher/files/file1.txt
Hello from ./launcher/files/file1.txt
Produced: ./launcher/files/file2.txt
Produced: ./launcher/files/file2.txt
Consumed: ./launcher/files/file2.txt
Consumed: ./launcher/files/file2.txt
Hello from ./launcher/files/file2.txt
Hello from ./launcher/files/file2.txt
Produced: ./launcher/files/file3.txt
Produced: ./launcher/files/file3.txt
Consumed: ./launcher/files/file3.txt
Hello from ./launcher/files/file3.txt
Consumed: ./launcher/files/file3.txt
Hello from ./launcher/files/file3.txt

Launching: ./launcher/build/app (run 3)
Produced: ./launcher/files/file1.txt
Produced: ./launcher/files/file1.txt
Consumed: ./launcher/files/file1.txt
Hello from ./launcher/files/file1.txt
Consumed: ./launcher/files/file1.txt
Hello from ./launcher/files/file1.txt
Produced: ./launcher/files/file2.txt
Produced: ./launcher/files/file2.txt
Consumed: ./launcher/files/file2.txt
Hello from ./launcher/files/file2.txt
Consumed: ./launcher/files/file2.txt
Hello from ./launcher/files/file2.txt
Produced: ./launcher/files/file3.txt
Produced: ./launcher/files/file3.txt
Consumed: ./launcher/files/file3.txt
Hello from ./launcher/files/file3.txt
Consumed: ./launcher/files/file3.txt
Hello from ./launcher/files/file3.txt

```

Рисунок 8.4 – Результат виконання програми

8.3 Висновки

Під час даної лабораторної роботи ми навчилися використовувати процеси та потоки при виконанні програм.