Párhuzamos programok

a)

A következő példaprogramban a fv függvény három példányban, egymással párhuzamosan fut. Az std::thread objektum a konstruktorában megkapja a futtatandó függvényt, valamint a függvény paraméterét/paramétereit. Miután elindította a fv függvényt a külön szálon, közben a main függvény futtatása tovább folytatódik, így indítja el a másik két szálat is, majd maga is elkezd kiírni, 'a' betűket. A szál objektum join tagfüggvénye addig vár, amíg a szál által futtatott függvény futása véget nem ér. A join után lehetünk biztosak benne, hogy az adott szál befejeződött, addig nem tudhatjuk, hogy hol tart éppen.

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

void fv(int i) {
    for (int j = 0; j < 100; ++j)
        std::cout << i;
}

int main() {
    std::thread t1{ fv, 1 };
    std::thread t2{ fv, 2 };
    std::thread t3{ fv, 3 };

    for (int j = 0; j < 100; ++j)
        std::cout << 'a';

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

Egy lehetséges futási eredmény:

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa11133a3aaaaa11222222223333311 1a1a1a1a1a1aa223311a12233332221133a3a333a3322a2111a1222a2a2a2331112233a3a3a3111a1122a23a311a1aa3a3a3a221131a13323aa2a223313333a322222a21113322a222a21113323a333a3a3a31121111a111aa3aa1a13322a2a23311a111a12223313aa221133a32212a21133a3a31a11111112232222331122331122322211111111111133311223311133221111333223323222323232323232323232222222

b)

Ha módosítjuk a programot, láthatjuk, hogy van egy kis probléma:

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

void fv(int i) {
    for (int j = 0; j < 20; ++j)
        std::cout << "Thread: " << i << std::endl;
}
```

```
int main() {
    std::thread t1{ fv, 1 };
    std::thread t2{ fv, 2 };
    std::thread t3{ fv, 3 };

    for (int j = 0; j < 20; ++j)
        std::cout << "Thread: " << 'a' << std::endl;

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

A kimenet valahogy így fog kinézni:

Thread: a
Thread: a
Thread: a
Thread: a
Thread: Thread: 3
Thread: 3
Thread: Thread: a
Thread: 21
3

Thread: 3Thread:
2
Thread: Thread: 31
Thread: Thread:
1Thread:
Thread: Thread: 12
aThread:
2
Thread: 31

Thread:
1
Thread: Thread: 2

Összekeveredtek a sorok, mert a szálak nem várják meg, hogy a másik szál által indított kiírás befejeződjön. Ez a probléma megoldható mutexek segítségével. Ha a mutex szabad, akkor az egy szál le tudja zárni. Ha közben egy másik szál is megpróbálja lezárni, akkor a mutex addig várakoztatja, míg a lezáró szál fel nem szabadítja:

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

std::mutex m;

void fv(int i) {
    for (int j = 0; j < 20; ++j) {
        m.lock();
```

```
        std::cout << "Thread: " << i << std::endl;
        m.unlock();
    }
}

int main() {
    std::thread t1{ fv, 1 };
    std::thread t2{ fv, 2 };
    std::thread t3{ fv, 3 };

    for (int j = 0; j < 20; ++j) {
        m.lock();
        std::cout << "Thread: " << 'a' << std::endl;
        m.unlock();
    }

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

```
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 2
Thread: 1
Thread: 1
```

```
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: 1
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: a
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
Thread: 3
```

Ez így már jó, de az látszik, hogy sokszor ugyanaz a szál fut egymás után. Ez azért van, mert a mutex nem teszi sorba a várakozó szálakat, hanem felszabadulásakor azt a szálat engedi lock-olni, amelyiket akarja, és közben a többi szál várakozik. Ez a program tehát hiába fut több szálon, valójában alapvetően

egy szálon működik, mert a kiírás teszi ki a futási idő nagy részét, ami nem párhuzamosítható. Valódi programok esetén nyilván számításokat teszünk a párhuzamos szálakba, nem kiírást, így az valóban ki tudja használni a többszálas processzort.

c)

Lock guard használatával automatizálhatjuk a mutex lezárását és felszabadítását. A következő program ugyanazt csinálja, mint az előző. A lock_guard a konstruktorában meghívja a paraméterként kapott mutex lock függvényét, a destruktorában pedig az unlock-ot. A destruktor a ciklusvégi }-nél fut le. Ha valaki kivételt dobna a ciklusban, a lokális változók akkor is megszűnnek, tehát automatikusan lefut a destruktor az unlock-kal, így nem marad lezárva a mutex.

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

std::mutex m;

void fv(int i) {
    for (int j = 0; j < 20; ++j) {
        std::lock_guard<std::mutex> guard{ m };
        std::cout << "Thread: " << i << std::endl;
    }
}

int main() {
    std::thread t1{ fv, 1 };
    std::thread t2{ fv, 2 };
    std::thread t3{ fv, 3 };

    for (int j = 0; j < 20; ++j) {
        std::lock_guard<std::mutex> guard{ m };
        std::cout << "Thread: " << 'a' << std::endl;
    }

    t1.join();
    t2.join();
    t3.join();

    return 0;
}
```

d)

Több mutex esetén lehet probléma, hogy két (vagy több) szál kölcsönösen egymásra vár. Mindegyik lezár egy mutexet, de csak akkor tudna továbbmenni, ha a másik által lezárt mutexet megkapná, de sosoem fogja, mert a másik pont az általa lezárt mutexre vár. Ez a holtpont (deadlock). Kielekulását mindig el kell kerülni megfelelő programtervezéssel. (Párhuzamos programnál, ha valaminek minimális az esélye, de nem 0, akkor az tuti be fog következni, tapasztalat.)

e)

Ha egy változóhoz kell több szálnak hozzáférnie, megtehetik mutexszel, de ennél gyorsabb, ha a változó atomi típusú. Az atomi változóhoz egyszerre csak egy folyamat férhet hozzá. Nem atomi változóra sose engedjünk rá több szálat, mert akár félig átírt változót is kaphat a másik szál.

```cpp
#include <atomic>

std::atomic<int> counter = 0;

void fv(int i) {
    counter++;
}
```

f)

condition_variable: várakoztatásra.