Další metody synchronizace

Na tomto cvičení byste si měli vyzkoušet další metody synchronizace vláken přes podmínkové proměnné (*condition variables*).

Domácí příprava

Nastudujte si použití podmínkových proměnných a k tomu příslušné funkce v knihovně `pthread`:

- `pthread_cond_init`
- `pthread_cond_destroy`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- `pthread_cond_wait`

Zadání úlohy

Vytvořte program simulující výrobní závod, který vyrábí 3 druhy výrobků. Každý výrobek musí projít pevně danou sekvencí operací na různých pracovištích. Pracoviště obsluhují dělníci, přičemž každý dělník je schopen obsluhovat právě jeden druh pracoviště. Počet dělníků a jemu příslušejících pracovišť je různý a v čase proměnlivý – dělníci přicházejí a odcházejí, pracoviště mohou přibývat nebo být vyřazována. Požadavky na výrobu, příchod a odchod dělníků a nákup/vyřazení strojů se zadávají přes standardní vstup aplikace následujícími příkazy (jeden příkaz na řádce ukončené '`\n`'):

- `make <výrobek>` – požadavek na výrobu výrobku; `<výrobek>` je "`A`", "`B`", nebo "`C`"

- `start <jméno> <pracoviště>` – příchod dělníka s uvedením jeho specializace
- `end <jméno>` – odchod dělníka
- `add <pracoviště>` – přidání nového pracoviště
- `remove <pracoviště>` – odebrání pracoviště

Parametry jsou odděleny mezerou. Při zadání neplatného příkazu či parametru, nebo špatného počtu parametrů, ignorujte celý řádek.

Celý proces je řízen dělníky. Každý dělník bude reprezentován samostatným vláknem, které bude vytvořeno při příchodu dělníka a ukončeno při jeho odchodu. Dělník potřebuje ke své práci polotovar (meziprodukt) a volné pracoviště pro které je specializován. Pokud nebude mít jedno nebo druhé, čeká. Pro první pracoviště nahrazuje meziprodukt požadavek na výrobu. Pokud má dělník vše potřebné k dispozici, vypíše informaci o své aktivitě a poté počká čas, který je definován pro každý typ operace. Formát výpisu aktivity je

```
<jméno> <pracoviště> <krok> <výrobek>
```

tedy např.

```
Karel vrtacka 2 A
```

Pokud dělník dokončí poslední operaci v procesu, vypíše

```
done <výrobek>
```

kde `<výrobek>` je kód výrobku `A`, `B`, nebo `C`.

Výrobní procesy pro výrobky `A` – `C` jsou následující:

```
A: 1:nuzky   - 2:vrtacka - 3:ohybacka  - 4:svarecka - 5:vrtacka - 6:lakovna
B: 1:vrtacka - 2:nuzky   - 3:freza     - 4:vrtacka  - 5:lakovna - 6:sroubovak
C: 1:freza   - 2:vrtacka - 3:sroubovak - 4:vrtacka  - 5:freza   - 6:lakovna
```

Časy operací v milisekundách na pracovištích jsou následující:

- `nuzky` : 100
- `vrtacka` : 200
- `ohybacka` : 150
- `svarecka` : 300

- `lakovna` : 400
- `sroubovak` : 250
- `freza` : 500

Můžete předpokládat, že nepřijdou dva dělníci stejného jména.

Pokud se odebírá pracoviště, odeberte přednostně neobsazené. Pokud není žádné pracoviště daného typu volné, vyřaďte libovolné, ovšem až po dokončení aktuální operace.

Dělník odchází z pracoviště buď při ukončování celé aplikace (viz níže) nebo jako reakce na příkaz `end`. Dělník který má odejít nejdříve dokončí aktuálně rozdělanou práci. Pokud má dělník odejít na základě příkazu `end`, nesmí vzít novou práci. Při odchodu dělníka ukončete jeho vlákno.

Pokud může dělník v danou chvíli pracovat na více různých místech, vybírá si místo s nejvyšším možným krokem tak, aby se prioritně zpracovávaly zakázky nejblíže dokončení. Je-li takových víc, vybírá mezi nimi výrobek blíže začátku abecedy, tedy např. výrobek `A` před `B`.

Uzavření standardního vstupu (tzn. Váš program načte `EOF` ze `stdin`) je požadavkem na ukončení celé aplikace, přičemž vlastní ukončení aplikace a odchod dělníků neukončených příkazem `end` nastane až v okamžiku, kdy žádný dělník nemůže pracovat (dělníci čekají a pokud se objeví práce začnou ještě pracovat). Před definitivním koncem aplikace ukončete vlákna všech dělníků a dealokujte alokovanou paměť. Návratový kód aplikace bude `0`.

Pokyny k implementaci

- Čekání všech vláken musí být efektivní, nesmí být vytěžován procesor (busy waiting).
- Nevytvářejte jiná vlákna než vlákna pro dělníky (byl by to problém pro evaluátor).
- Jsou-li dostupné zdroje, dělník musí započít práci okamžitě (co nejrychleji, bez zbytečné prodlevy).
- Standardní vstup čtěte pouze z hlavního vlákna.
- Příkazy ze vstupu vykonávejte bez zbytečného odkladu, tj. hlavní vlákno neblokujte, pokud to není nezbytně nutné. Zejména se nesnažte v hlavním vlákně čekat na to, až nějaký dělník něco provede. Cílem je, aby hlavní vlákno nebylo blokováno a mohlo tak rychle reagovat na zaslané příkazy.
- Standardní chybový výstup můžete použít pro libovolné ladící výpisy.
- Výpis aktivity musí být proveden tak, aby nemohlo dojít k prohození s výpisem jiné aktivity, která začala později.

C/C++ Rust

- Binární soubor aplikace se bude jmenovat `factory` a bude vytvořen ve stejném adresáři, kde se nachází `Makefile`.
- Program překládejte s příznaky `-Wall -g -O2` a navíc s příznaky v proměnné `EXTRA_CFLAGS`. Pokud tato proměnná není definována na příkazové řádce `make`, nastavte její hodnotu na "`-fsanitize=address -fno-omit-frame-pointer`" (viz např. operátor `?=`). Pokud provádíte překlad a linkování odděleně, používejte příznaky v `EXTRA_CFLAGS` také při linkování.
- Pro načítání vstupu můžete použít následující vzor: (Případně alternativní šablonu).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* You can use these functions and data structures to transform strin
 * numbers and use them in arrays
 */
enum place {
    NUZKY, VRTACKA, OHYBACKA, SVARECKA, LAKOVNA, SROUBOVAK, FREZA,
    _PLACE_COUNT
};

const char *place_str[_PLACE_COUNT] = {
    [NUZKY] = "nuzky",
    [VRTACKA] = "vrtacka",
    [OHYBACKA] = "ohybacka",
    [SVARECKA] = "svarecka",
    [LAKOVNA] = "lakovna",
    [SROUBOVAK] = "sroubovak",
    [FREZA] = "freza",
};

enum product {
    A, B, C,
    _PRODUCT_COUNT
};
```

```
const char *product_str[_PRODUCT_COUNT] = {
    [A] = "A",
    [B] = "B",
    [C] = "C",
};

int find_string_in_array(const char **array, int length, char *what)
{
    for (int i = 0; i < length; i++)
        if (strcmp(array[i], what) == 0)
            return i;
    return -1;
}

/* It is not necessary to represent each working place with a dynamic
 * allocated object.  You can store only number of ready places
 *
 * int ready_places[_PLACE_COUNT];
 */

/* It is not necessary to represent each part as a dynamically alloca
 * object.  you can have only number of parts for each working phase
 *
 * #define _PHASE_COUNT 6
 * int parts[_PRODUCT_COUNT][_PHASE_COUNT]
 */

int main(int argc, char **argv)
{
    /* Initialize your internal structures, mutexes and conddition va
     */
    while (1) {
        char *line, *cmd, *arg1, *arg2, *arg3, *saveptr;
        int s = scanf(" %m[^\n]", &line);
        if (s == EOF)
            break;
        if (s == 0)
            continue;

        cmd  = strtok_r(line, " ", &saveptr);
```

```
        arg1 = strtok_r(NULL, " ", &saveptr);
        arg2 = strtok_r(NULL, " ", &saveptr);
        arg3 = strtok_r(NULL, " ", &saveptr);

        if (strcmp(cmd, "start") == 0 && arg1 && arg2 && !arg3) {
            /* - start new thread for new worker
             * - copy (e.g. strdup()) worker name from arg1, the
             *   arg1 will be removed at the end of scanf cycle
             * - workers should have dynamic objects, you don't know
             *   total number of workers
             */
        } else if (strcmp(cmd, "make") == 0 && arg1 && !arg2) {
            /* int product = find_string_in_array(
             *     product_str,
             *     _PRODUCT_COUNT,
             *     arg1
             * );
             *
             * if (product >= 0) {.....
             *     add the part to factory cycle
             *     you need to wakeup worker to start working if poss
             *     ...
             *     }
             */
        } else if (strcmp(cmd, "end") == 0 && arg1 && !arg2) {
            /* tell the worker to finish
             * the worker has to finish their work first
             * you should not wait here for the worker to finish
             *
             * if the worker is waiting for work
             * you need to wakeup the worker
             */
        } else if (strcmp(cmd, "add") == 0 && arg1 && !arg2) {
            /* add new place
             *
             * if worker and part is ready, start working - wakeup wo
             */
        } else if (strcmp(cmd, "remove") == 0 && arg1 && !arg2) {
            /* if you cannot remove empty place you cannot wait for f
             * work
```

```
                 */
            } else {
                fprintf(stderr, "Invalid command: %s\n", line);
            }
            free(line);
        }

        /* Wait for every worker to finish their work. Nobody should be a
         * continue.
         */
    }
```

Ukázkový vstup a odpovídající výstup

Vstup

```
add nuzky
add vrtacka
add ohybacka
add svarecka
add lakovna
add sroubovak
add freza

start Nora nuzky
start Vojta vrtacka
start Otakar ohybacka
start Sofie svarecka
start Lucie lakovna
start Stepan sroubovak
start Filip freza

make A
```

Výstup

```
Nora nuzky 1 A
Vojta vrtacka 2 A
Otakar ohybacka 3 A
Sofie svarecka 4 A
Vojta vrtacka 5 A
Lucie lakovna 6 A
done A
```

Domácí příprava na další cvičení

Na přištím tyden: žádná – na vypracování této úlohy máte 2 týdny.

Za 2 týdny: Seznamte se se základními instrukcemi architektury x86 a způsobem, jakým vkládat instrukce assembleru přímo do zdrojového kódu v jazyce C/C++.