6. Další metody synchronizace

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 (např. neexistující pracoviště), nebo špatného počtu parametrů, ignorujte celý řádek. Doporučujeme informovat o tom uživatele výpisem do standardního chybového výstupu.

Výrobní 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:

	1	2	3	4	5	6
A	nuzky	vrtacka	ohybacka	svarecka	vrtacka	lakovna
B	vrtacka	nuzky	freza	vrtacka	lakovna	sroubovak

	1	2	3	4	5	6
C	freza	vrtacka	sroubovak	vrtacka	freza	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 dělník vypíše na standardní výstup:

```
<jméno> goes home
```

a poté ukončí své 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. Po tomto požadavku dělník odchází v okamžiku, kdy je jisté, že jeho profese už nebude potřeba, tj. pokud nikdo dané profese momentálně nepracuje a ani pracovat nebude moct, např. protože není k dispozici žádné pracoviště pro jeho profesi, nebo když se už žádný polotovar nemůže dostat na pracoviště jeho profese.

> **Příklad ukončování aplikace:** Uvažujme, že jsou k dispozici všechna pracoviště, pro každou profesi je k dispozici alespoň jeden dělník a že jsou při příjmu `EOF`

> rozpracovány pouze výrobky `B` a `C`. Okamžitě tedy odchází ohýbači a svářeči (ti mohou pracovat jen na `A`). Ostatní odcházejí podle fáze zpracování výrobků `B` a `C`. Např. lakýrníci odcházejí, když je dokončen poslední výrobek `C` a všechny výrobky `B` mají dokončenou 5. fázi.

Před definitivním koncem aplikace ukončete vlákna všech dělníků a dealokujte dynamicky 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.
- Nesnažte se optimalizovat až příliš, pokud by to významně zkomplikovalo program. Např. je přípustné dělníka vzbudit, i když dotyčný zjistí, že pro něj není práce a opět se uspí.
- 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 str
 * 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 dynam
 * allocated object.  You can store only numbers of ready places.
 *
 * int ready_places[_PLACE_COUNT];
 */

/* It is not necessary to represent each part as a dynamically allo
 * object.  You can store only a number of parts for each working p
 *
 * #define _PHASE_COUNT 6
 * int parts[_PRODUCT_COUNT][_PHASE_COUNT]
 */

int main(int argc, char **argv)
{
    /* Initialize your internal structures, mutexes and conddition
     */
    char *line = NULL;
    size_t sz = 0;
    while (1) {
        char *cmd, *arg1, *arg2, *arg3, *saveptr;

        if (getline(&line, &sz, stdin) == -1)
            break; /* Error or EOF */
```

```
cmd = strtok_r(line, " \r\n", &saveptr);
arg1 = strtok_r(NULL, " \r\n", &saveptr);
arg2 = strtok_r(NULL, " \r\n", &saveptr);
arg3 = strtok_r(NULL, " \r\n", &saveptr);

if (!cmd) {
    continue; /* Empty line */
} else if (strcmp(cmd, "start") == 0 && arg1 && arg2 && !ar
    /* - 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 kno
     *   total number of workers
     */
} else if (strcmp(cmd, "make") == 0 && arg1 && !arg2) {
    int product = find_string_in_array(product_str, _PRODUC

    if (product >= 0) {
        /* add the part to factory cycle */
        /* you need to wakeup a worker to start working if
    }
} 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 a new place
     *
     * if worker and part is ready, start working - wakeup
     */
} else if (strcmp(cmd, "remove") == 0 && arg1 && !arg2) {
    /* If no place is currently idle, queue removal for lat
     * You should not wait here until the place becomes idl
     */
```

```
            } else {
                fprintf(stderr, "Invalid command: %s\n", line);
            }
        }
        free(line);
        /* Wait for every worker to finish their work. Nobody should be
         * continue.
         */
    }
```

- Pro kontrolu správného používání mutexů a detekci potenciálních chyb souběhu (race conditions) můžete použít thread sanitizer. Pokud máte `Makefile` napsaný podle pokynů výše, mělo by stačit přeložit program pomocí:

  ```
  make EXTRA_CFLAGS=-fsanitize=thread
  ```

  Když pak spustíte váš program normálním způsobem, thread sanitizer vás bude upozorňovat na problémy pomocí výpisů podobných tomu níže:

  ```
  WARNING: ThreadSanitizer: data race (pid=366626)
    Write of size 1 at 0x0000004050f8 by main thread:
      #0 main factory.c:297 (factory+0x401489)

    Previous read of size 1 at 0x0000004050f8 by thread T5 (mutexes:
      #0 is_proffession_needed factory.c:124 (factory+0x401a28)
      #1 worker factory.c:142 (factory+0x401ad7)

    Location is global 'factory' of size 304 at 0x0000004050a0 (facto

    Mutex M0 (0x0000004050a0) created at:
      #0 pthread_mutex_init <null> (libtsan.so.2+0x42288)
      #1 main factory.c:261 (factory+0x401216)

    Thread T5 (tid=366703, running) created by main thread at:
      #0 pthread_create <null> (libtsan.so.2+0x41cd6)
      #1 start_worker factory.c:212 (factory+0x40215e)
      #2 main factory.c:282 (factory+0x40157d)

  SUMMARY: ThreadSanitizer: data race factory.c:297 in main
  ```

Z výpisu je vidět, že okolo řádku 297 jsme zapomněli zamknout mutex M0, který byl vytvořen na řádku 261.

Ukázkové vstupy a odpovídající výstupy

Výroba A

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
```

Nedokončený výrobek a odchod

```
$ ./factory <<EOF
start Nora nuzky
add nuzky
make A
EOF

Nora nuzky 1 A
Nora goes home
```

Rozdílný odchod dělníků

```
$ ./factory <<EOF
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
make C
EOF

Nora nuzky 1 A
Filip freza 1 C
Nora goes home
Vojta vrtacka 2 A
```

```
Otakar ohybacka 3 A
Otakar goes home
Sofie svarecka 4 A
Vojta vrtacka 2 C
Stepan sroubovak 3 C
Sofie goes home
Vojta vrtacka 5 A
Stepan goes home
Vojta vrtacka 4 C
Lucie lakovna 6 A
Vojta goes home
Filip freza 5 C
done A
Filip goes home
Lucie lakovna 6 C
done C
Lucie goes home
```

Domácí příprava na další cvičení

Na příští týden: 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++.