Posix threads

> GitHub-classroom для самостоятельно изучающих курс **не** в рамках университетских программ

Ваша задача - реализовать классический паттерн producer-consumer с небольшими дополнительными условиями. Программа должна состоять из 3+N потоков:

1. Главный
2. producer
3. interruptor
4. N потоков consumer

В файл, чтение которого уже реализовано в шаблоне кода в блокирующкем режиме, пишется список чисел, разделённых пробелом (читать можно до переноса строки). Длина списка чисел не задаётся - считывание происходит до перевода каретки.

- Задача producer-потока - получить на вход список чисел, и по очереди использовать каждое значение из этого списка для обновления переменной разделяемой между потоками
- Задача consumer-потоков отреагировать на уведомление от producer и набирать сумму полученных значений. Также этот поток должен защититься от попыток потока-interruptor его остановить. Дополнительные условия:
    1. Функция, исполняющая код этого потока consumer_routine, должна принимать указатель на объект/переменную, из которого будет читать обновления
    2. После суммирования переменной поток должен заснуть на случайное количество миллисекунд, верхний предел будет передан на вход приложения (0 миллисекунд также должно корректно обрабатываться). Вовремя сна поток не должен мешать другим потокам consumer выполнять свои задачи, если они есть
    3. Потоки consumer не должны дублировать вычисления друг с другом одних и тех же значений
    4. В качестве возвращаемого значения поток должен вернуть свою частичную посчитанную сумму
- Задача потока-interruptor проста: пока происходит процесс обновления значений, он должен постоянно пытаться остановить случайный поток consumer (вычисление случайного потока происходит перед каждой попыткой остановки). Как только поток producer произвел последнее обновление, этот поток завершается. В этом потоке можно выполнять вспомогательные действия которые помогут корректно обработать сигнал
- Функция run_threads должна запускать все потоки, дожидаться их выполнения, и возвращать результат общего суммирования
- Завершение приложения происходит или при считывании перевода каретки из файла или по посылке сигнала SIGTERM, обработку которого нужно также добавить. В обработчике сигнала можно вызывать только signal-safe функции https://man7.org/linux/man-pages/man7/signal-safety.7.html В случае, если в этот момент поток, читающий данные с файла, находится в режиме блокирующего чтения, - он также должен корректно завершиться. Полезно присмотреться к std::sig_atomic_t - позволяет потокобезопасно и signal-safe обращаться к переменной такого типа.
- При сигнале в качестве вывода нужно выдавать посчитанную на этот момент сумму

Архитектурные ограничения:

1. В нашем producer/consumer необходимо реализоваться потоковую обработку данных для этих целей запрещается загружать весь файл в ОЗУ, иначе на больших файлах обработка не будет помещаться в ограничения ОЗУ на вычислительном устройстве.
2. При отсутствии входных данных ядра процессора не должны вхолостую есть такты

Для обеспечения межпоточного взаимодействия допускается использование только pthread API. На вход приложения передаётся 2 аргумента при старте именно в такой последовательности:

1. Число потоков consumer
2. Верхний предел сна consumer в миллисекундах

В поток вывода должно попадать только результирующее значение, по умолчанию никакой отладочной или запросной информации выводиться не должно. В случае детектирования ошибок нужно выдавать не нулевой код возврата.

```
#include <iostream>
#include <fstream>
#include <pthread.h>

void* producer_routine(void* arg) {
  // Wait for consumer to start.
  // You should use this waiting only for debugging your code
  // For the final solution please remove this waiting

  // Read data, loop through each value and update the value, notify
consumer, wait for consumer to process
  std::ifstream ifs("in.txt");
  // ...
}

void* consumer_routine(void* arg) {
  // notify about start
  // you should use this notification only for debugging your code
  // for the final solution please remove this notification


  // for every update issued by producer, read the value and add to sum
  // return pointer to result (for particular consumer)
}

void* consumer_interruptor_routine(void* arg) {
  // wait for consumers to start
  // you should use this waiting only for debugging your code
  // for the final solution please remove this waiting

  // interrupt random consumer while producer is running
}

int run_threads() {
```

```
  // start N threads and wait until they're done
  // return aggregated sum of values

  return 0;
}

int main() {
    std::cout << run_threads() << std::endl;
    return 0;
}
```

From:
http://wiki.osll.ru/ - **Open Source & Linux Lab**

Permanent link:
http://wiki.osll.ru/doku.php/courses:high_performance_computing:producer_consumer?rev=1745725878

Last update: **2025/04/27 06:51**