Технологии параллельного программирования на C++

Семинар 4

Условная переменная

- Условная переменная в Pthreads – это переменная типа pthread_cond_t, которая обеспечивает блокирование одного или нескольких потоков до тех пор, пока не придёт сигнал, или не пройдёт максимально установленное время ожидания. Условная переменная используется совместно с ассоциированным с ней мьютексом.
- Перед использованием необходимо инициализировать условную переменную:

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

Или с помощью макроса **PTHREAD_COND_INITIALIZER**

- Для уничтожения условной переменной используется функция:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Для ожидания сигнала можно использовать функцию, которая ждет неопределенно долго:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Условная переменная

- Чтобы разблокировать определенный поток, заблокированной с помощью условной переменной cond, используется функция:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- Все блокированные потоки можно разблокировать функцией:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Барьер для N потоков

*псевдокод

```
#define SYNC_MAX_COUNT 10
void SynchronizationPoint() {
 static mutex_t sync_lock = PTHREAD_MUTEX_INITIALIZER;
 static cond_t sync_cond = PTHREAD_COND_INITIALIZER;
 static int sync_count = 0;

 /* блокировка доступа к счетчику */
 pthread_mutex_lock(&sync_lock);
 sync_count++;

 /* проверка: следует ли продолжать ожидание */
 if (sync_count < SYNC_MAX_COUNT)
    pthread_cond_wait(&sync_cond, &sync_lock);
 else
 /* оповестить о достижении данной точки всеми */
    pthread_cond_broadcast(&sync_cond);

 /* активизация взаимной блокировки - в противном случае
 из процедуры сможет выйти только одна нить! */
 pthread_mutex_unlock(&sync_lock);
}
```

Условные переменные

- Нужны как механизм взаимодействия потоков, в отличие от мьютексов
- Всегда используется с мьютексом
- Предназначена для уведомления событии
- Атомарно освобождает мьютекс при wait()
- Хорошо подходит для задач типа «производитель-потребитель»

Задание

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

- главный
- producer
- interruptor
- N потоков consumer

На вход приложения передаётся 2 аргумента при старте именно в такой последовательности: число потоков consumer и верхний предел сна consumer в миллисекундах

На стандартный ввод программе подается строка - список чисел, разделённых пробелом (читать можно до конца ввода). Длина списка чисел не задаётся - считывание происходит до перевода каретки. Программа должна вывести сумму этих чисел.

Задание (продолжение)

- Функция run_threads должна запускать все потоки, дожидаться их выполнения, и возвращать результат общего суммирования.

```
int run_threads() {
  // start N threads and wait until they're done
  // return aggregated sum of values

  return 0;
}
```

Задание (продолжение)

- Задача producer-потока - получить на вход список чисел, и по очереди использовать каждое значение из этого списка для обновления переменной разделяемой между потоками

```
void* producer_routine(void* arg) {
  // Wait for consumer to start

  // Read data, loop through each value and update the value, notify consumer, wait for consumer to process
}
```

Задание (продолжение)

- 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

```
void* consumer_routine(void* arg) {
  // notify about start
  // for every update issued by producer, read the value and add to sum
  // return pointer to result (for particular consumer)
}
```

Задание (продолжение)

- 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.

```
void* consumer_interruptor_routine(void* arg) {
  // wait for consumers to start

  // interrupt random consumer while producer is running
}
```

Параметры функции **main**