OS2 Lab13

checkbox	☑
due date	@October 15, 2021
class	OS
Status	approved

Task

Решите задачу 11 с использованием условной переменной и минимально необходимого количества мутексов.

Notes

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <stdbool.h>

pthread_mutex_t mutex;
pthread_cond_t conditionalVariable;

bool parentPrinted = false;

void* threadRoutine(void* args){
    for(int i = 0; i < 10; i++){
        pthread_mutex_lock(&mutex);
        while (!parentPrinted){
            pthread_cond_wait(&conditionalVariable, &mutex);
        }
        printf("child\n");
        parentPrinted = false;
        pthread_cond_signal(&conditionalVariable);
        pthread_mutex_unlock(&mutex);
    }
}

int main(){
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&conditionalVariable, NULL);

    pthread_t childThread;

    if (pthread_create(&childThread, NULL, threadRoutine,
        perror("pthread_create:");
    }

    for(int i = 0; i < 10; i++){
        pthread_mutex_lock(&mutex);
        while (parentPrinted){
            pthread_cond_wait(&conditionalVariable, &mutex);
        }
        printf("parent\n");
        parentPrinted = true;
        pthread_cond_signal(&conditionalVariable);
        pthread_mutex_unlock(&mutex);
    }

    pthread_mutex_destroy(&mutex);

    pthread_exit(0);
}
```

Ложные пробуждения:

> When using condition variables there is always a boolean predicate, an invariant, associated with each condition wait that must be true before the thread should proceed. Spurious wakeups from the **pthread_cond_wait()**, **pthread_cond_timedwait()**, or **pthread_cond_reltimedwait_np()** functions could occur. Since the return from **pthread_cond_wait()**, **pthread_cond_timedwait()**, or **pthread_cond_reltimedwait_np()** does not imply anything about the value of this predicate, the predicate should always be reevaluated.

Spurious wakeup

From Wikipedia, the free encyclopedia

This article's **tone or style may not reflect the encyclopedic tone used on Wikipedia**. See Wikipedia's guide to writing better articles for suggestions. *(September 2020)* *(Learn how and when to remove this template message)*

A **spurious wakeup** happens when a thread wakes up from waiting on a condition variable that's been signaled, only to discover that the condition it was waiting for isn't satisfied. It's called spurious because the thread has seemingly been awakened for no reason. But spurious wakeups don't happen for no reason: they usually happen because, in between the time when the condition variable was signaled and when the waiting thread finally ran, another thread ran and changed the condition. There was a race condition between the threads, with the typical result that sometimes, the thread waking up on the condition variable runs first, winning the race, and sometimes it runs second, losing the race.

On many systems, especially multiprocessor systems, the problem of spurious wakeups is exacerbated because if there are several threads waiting on the condition variable when it's signaled, the system may decide to wake them all up, treating every `signal( )` to wake one thread as a `broadcast( )` to wake all of them, thus breaking any possibly expected 1:1 relationship between signals and wakeups.[1] If there are ten threads waiting, only one will win and the other nine will experience spurious wakeups.

To allow for implementation flexibility in dealing with error conditions and races inside the operating system, condition variables may also be allowed to return from a wait even if not signaled, though it is not clear how many implementations actually do that. In the Solaris implementation of condition variables, a spurious wakeup may occur without the condition being signaled if the process is signaled; the wait system call aborts and returns `EINTR`.[2] The Linux pthread implementation of condition variables guarantees it will not do that.[3][4]

Because spurious wakeups can happen whenever there's a race and possibly even in the absence of a race or a signal, when a thread wakes on a condition variable, it should always check that the condition it sought is satisfied. If it's not, it should go back to sleeping on the condition variable, waiting for another opportunity.

Interrupted Waits on Condition Variables

When an unmasked, caught signal is delivered to a thread waiting on a condition variable, the thread returns from the signal handler with a spurious wakeup. A spurious wakeup is a wakeup not caused by a condition signal call from another thread. In this case, the Solaris threads interfaces, `cond_wait()` and `cond_timedwait()`, return `EINTR` while the POSIX threads interfaces, `pthread_cond_wait()` and `pthread_cond_timedwait()`, return 0. In all cases, the associated mutex lock is reacquired before returning from the condition wait.

Reacquition of the associated mutex lock does not imply that the mutex is locked while the thread is executing the signal handler. The state of the mutex in the signal handler is undefined.

The implementation of `libthread` in releases of the Solaris software prior to the Solaris 9 release guaranteed that the mutex was held while in the signal handler. Applications that rely on this old behavior require revision for the Solaris 9 release and subsequent releases.

The following explanation is given by David R. Butenhof in "Programming with POSIX Threads" (p. 80):

> Spurious wakeups may sound strange, but on some multiprocessor systems, making condition wakeup completely predictable might substantially slow all condition variable operations.

In the following comp.programming.threads discussion, he expands on the thinking behind the design:

```
Patrick Doyle wrote:
> In article , Tom Payne   wrote:
> >Kaz Kylheku  wrote:
> >: It is so because implementations can sometimes not avoid inserting
> >: these spurious wakeups; it might be costly to prevent them.

> >But why?  Why is this so difficult?  For example, are we talking about
> >situations where a wait times out just as a signal arrives?

> You know, I wonder if the designers of pthreads used logic like this:
> users of condition variables have to check the condition on exit anyway,
> so we will not be placing any additional burden on them if we allow
> spurious wakeups; and since it is conceivable that allowing spurious
> wakeups could make an implementation faster, it can only help if we
> allow them.

> They may not have had any particular implementation in mind.

You're actually not far off at all, except you didn't push it far enough.

The intent was to force correct/robust code by requiring predicate loops. This was
driven by the provably correct academic contingent among the "core threadies" in
the working group, though I don't think anyone really disagreed with the intent
once they understood what it meant.

We followed that intent with several levels of justification. The first was that
"religiously" using a loop protects the application against its own imperfect
coding practices. The second was that it wasn't difficult to abstractly imagine
machines and implementation code that could exploit this requirement to improve
the performance of average condition wait operations through optimizing the
synchronization mechanisms.
```

Условные переменные

Условная переменная представляет собой примитив синхронизации. Над ней определены две основные операции: `wait` и `signal`. Нить, выполнившая операцию `wait`, блокируется до того момента, пока другая нить не выполнит операцию `signal`. Таким образом, операцией `wait` первая нить сообщает системе, что она ждет выполнения какого-то условия, а операцией `signal` вторая нить сообщает первой, что параметры, от которых зависит выполнение условия, возможно, изменились.

Основное применение условных переменных - это сценарий производитель-потребитель. Рассмотрим две нити, одна из которых генерирует данные,а другая - перерабатывает их. В простейшем случае производитель помещает каждую следующую порцию данных в разделяемую переменную, а потребитель считывает ее оттуда. При этом могут возникать две проблемы. Если производитель работает быстрее потребителя, то он может записать очередную порцию данных до того, как потребитель прочитает предыдущую. При этом предыдущая порция данных будет потеряна. Если же потребитель работает быстрее производителя, он может обработать одну и ту же порцию данных несколько раз. Легко понять, что при помощи одного мутекса эти проблемы устранить невозможно. Несколько сложнее понять, что их нельзя решить и при помощи двух мутексов (в качестве упражнения слушателю предлагается доказать это утверждение). Даже при использовании трех мутексов мы будем вынуждены использовать для начальной синхронизации холостой цикл или какой-то другой примитив межпоточного взаимодействия.

Точное описание действия функции `pthread_cond_wait(3C)` звучит так. Эта функция имеет два параметра, `pthread_cond_t *cond` и `pthread_mutex_t *mx`. При вызове `wait` мутекс должен быть захвачен, в противном случае результат не определен. `Wait` освобождает мутекс и блокирует нить до момента вызова другой нитью `pthread_cond_signal`. После пробуждения `wait` пытается захватить мутекс; если это не получается, он блокируется до того момента, пока мутекс не освободят.

Мутекс используется для защиты данных, используемых при вычислении условия, с которым связана наша условная переменная. Условие необходимо проверять как перед вызовом `pthread_cond_wait(3C)`, так и после выхода из этой функции. Проверка условия перед вызовом позволяет защититься от так называемой "ошибки потерянного пробуждения" (lost wakeup), т.е. от