**Приоритетное планирование**

Приоритетное планирование

1. Описание функций для управления планированием

POSIX-2001 предоставляет набор функций для работы с политиками планирования процессов и потоков управления. Например:

sched\_getscheduler() — возвращает текущую политику планирования процесса.

sched\_getparam() — возвращает параметры планирования процесса (приоритеты).

sched\_setscheduler() — устанавливает политику и параметры планирования для процесса.

sched\_setparam() — устанавливает только параметры планирования (например, приоритет).

sched\_get\_priority\_min() и sched\_get\_priority\_max() — возвращают минимальные и максимальные значения приоритетов для указанной политики.

sched\_rr\_get\_interval() — возвращает величину кванта процессорного времени для политики циклического планирования (SCHED\_RR).

sched\_yield() — позволяет потоку добровольно уступить процессор.

Эти функции позволяют управлять приоритетами процессов и потоков, а также влиять на их порядок выполнения в многозадачных системах.

2. Пример использования функций планирования

Ниже приведен пример программы, которая демонстрирует использование функций для опроса политик планирования и изменения атрибутов процесса:

#include <stdio.h>

#include <sched.h>

int main(void) {

    struct sched\_param shdprm;   // Параметры планирования

    struct timespec interval;    // Квант процессорного времени

    // Вывод допустимых диапазонов приоритетов для различных политик планирования

    printf("Допустимые диапазоны приоритетов:\n");

    printf("SCHED\_FIFO : от %d до %d\n", sched\_get\_priority\_min(SCHED\_FIFO), sched\_get\_priority\_max(SCHED\_FIFO));

    printf("SCHED\_RR: от %d до %d\n", sched\_get\_priority\_min(SCHED\_RR), sched\_get\_priority\_max(SCHED\_RR));

    printf("SCHED\_OTHER: от %d до %d\n", sched\_get\_priority\_min(SCHED\_OTHER), sched\_get\_priority\_max(SCHED\_OTHER));

    // Опрос текущей политики планирования

    printf("Текущая политика планирования:\n");

    switch (sched\_getscheduler(0)) {

        case SCHED\_FIFO:

            printf("SCHED\_FIFO\n");

            break;

        case SCHED\_RR:

            printf("SCHED\_RR\n");

            break;

        case SCHED\_OTHER:

            printf("SCHED\_OTHER\n");

            break;

        default:

            printf("Неизвестная политика планирования\n");

    }

    // Получение текущего приоритета процесса

    if (sched\_getparam(0, &shdprm) == 0) {

        printf("Текущий приоритет: %d\n", shdprm.sched\_priority);

    }

    // Установка новой политики и приоритета

    shdprm.sched\_priority = 50;

    if (sched\_setscheduler(0, SCHED\_RR, &shdprm) == -1) {

        perror("Ошибка при установке планирования");

    }

    // Опрос кванта процессорного времени для политики SCHED\_RR

    if (sched\_rr\_get\_interval(0, &interval) == 0) {

        printf("Квант процессорного времени: %g сек\n", interval.tv\_sec + interval.tv\_nsec / 1000000000.0);

    }

    return 0;

}

Объяснение программы:

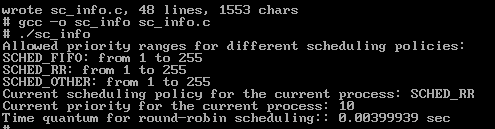
Вывод диапазонов приоритетов для разных политик планирования (SCHED\_FIFO, SCHED\_RR, SCHED\_OTHER) с помощью функций sched\_get\_priority\_min() и sched\_get\_priority\_max().

Опрос текущей политики планирования через sched\_getscheduler(). Это функция возвращает одну из трёх политик: SCHED\_FIFO, SCHED\_RR или SCHED\_OTHER.

Получение текущего приоритета с помощью sched\_getparam().

Изменение политики планирования на SCHED\_RR с установкой нового приоритета (50) через sched\_setscheduler().

Опрос величины кванта процессорного времени для политики SCHED\_RR через sched\_rr\_get\_interval().



3. Пример инверсии приоритетов

Пример моделирует ситуацию инверсии приоритетов, когда поток с низким приоритетом захватывает общий ресурс (семафор), но не может освободить его из-за того, что поток со средним приоритетом блокирует процессор. В это время поток с высоким приоритетом также хочет получить доступ к ресурсу, но он оказывается заблокированным.

#include <stdio.h>

#include <pthread.h>

#include <sched.h>

#include <semaphore.h>

#include <assert.h>

static sem\_t sem\_mi;  // Семафор для среднего приоритета

static sem\_t sem\_hi;  // Семафор для высокого приоритета

static double s = 0;

void \*start\_hi(void \*dummy) {

    printf("Поток с высоким приоритетом перед захватом семафора\n");

    assert(sem\_wait(&sem\_hi) == 0);

    printf("Поток с высоким приоритетом после захвата семафора\n");

    assert(sem\_post(&sem\_hi) == 0);

    return NULL;

}

void \*start\_mi(void \*dummy) {

    int i;

    double d = 1;

    printf("Поток со средним приоритетом перед захватом семафора\n");

    assert(sem\_wait(&sem\_mi) == 0);

    for (i = 1; i < 100000000; i++) {  // Долгие вычисления

        s += d / i;

        d = -d;

    }

    printf("Поток со средним приоритетом перед освобождением семафора\n");

    assert(sem\_post(&sem\_mi) == 0);

    return &s;

}

int main(void) {

    pthread\_t pt\_mi, pt\_hi;

    pthread\_attr\_t attrob;

    struct sched\_param shdprm;

    assert(sem\_init(&sem\_mi, 0, 0) == 0);

    assert(sem\_init(&sem\_hi, 0, 0) == 0);

    // Устанавливаем текущий поток с низким приоритетом

    shdprm.sched\_priority = sched\_get\_priority\_max(SCHED\_FIFO) - 31;

    assert(pthread\_setschedparam(pthread\_self(), SCHED\_FIFO, &shdprm) == 0);

    // Инициализируем атрибуты потоков

    assert(pthread\_attr\_init(&attrob) == 0);

    assert(pthread\_attr\_setinheritsched(&attrob, PTHREAD\_EXPLICIT\_SCHED) == 0);

    assert(pthread\_attr\_setschedpolicy(&attrob, SCHED\_FIFO) == 0);

    // Поток со средним приоритетом

    shdprm.sched\_priority += 15;

    assert(pthread\_attr\_setschedparam(&attrob, &shdprm) == 0);

    assert(pthread\_create(&pt\_mi, &attrob, start\_mi, NULL) == 0);

    // Поток с высоким приоритетом

    shdprm.sched\_priority += 15;

    assert(pthread\_attr\_setschedparam(&attrob, &shdprm) == 0);

    assert(pthread\_create(&pt\_hi, &attrob, start\_hi, NULL) == 0);

    // Освобождение семафоров для обоих потоков

    assert(sem\_post(&sem\_mi) == 0);

    assert(sem\_post(&sem\_hi) == 0);

    pthread\_join(pt\_mi, NULL);

    pthread\_join(pt\_hi, NULL);

    assert(sem\_destroy(&sem\_mi) == 0);

    assert(sem\_destroy(&sem\_hi) == 0);

    return 0;

}

Поток с низким приоритетом захватывает ресурс (семафор), но поток со средним приоритетом начинает занимать процессор, не давая низкоприоритетному потоку освободить семафор. Это блокирует поток с высоким приоритетом, который также хочет получить доступ к ресурсу.

Решение инверсии приоритетов заключается в использовании протокола наследования приоритетов, который временно повышает приоритет потока с низким приоритетом, чтобы он мог завершить свою работу и освободить ресурс.

