МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. Шухова»
(БГТУ им. В. Г. Шухова)

Кафедра программного обеспечения вычислительной
техники и автоматизированных систем

Лабораторная работа №2
по дисциплине: «Операционные системы»
на тему: «Процессы и потоки в ОС Linux (Ubuntu): сравнение, механизмы синхронизации. Парадигмы межпроцессорного взаимодействия.»

Выполнил: ст. группы ПВ-223
Дмитриев Андрей Александрович

Проверили:
доц. Островский Алексей Мичеславович,
асс. Четвертухин Виктор Романович

Белгород, 2024

Цель работы: Изучить различия между процессами и потоками в ОС Linux (Ubuntu), а также освоить механизмы синхронизации и межпроцессорного взаимодействия для обеспечения корректной работы программ в многозадачной среде.

Условие индивидуального задания:

Змей Горыныч имеет три головы, каждая из которых независимо от других голов ест продукцию с двух кондитерских фабрик. Каждая фабрика производит разные типы кондитерских изделий (торты, пирожные, конфеты, пряники) с различной скоростью. Головам нужно получать продукцию из общего склада, но склад ограничен по объему. Задача — организовать взаимодействие между фабриками (производителями) и головами Змея Горыныча (потребителями) так, чтобы они корректно синхронизировались при производстве и потреблении продукции, избегая конфликтов, минимизируя ситуации простоя и переполнения склада.

Для реализации склада использовать, в случае процессов, именованный канал FIFO (mkfifo), для реализации потоков — очередь с блокировкой. Для синхронизации, в случае процессов, использовать семафоры (sys/sem.h), а для потоков — futex (linux/futex.h, sys/syscall.h).

Ход выполнения работы

Задание 1.

Текст программы на языке C с комментариями.

```c
#define QUEUE_SIZE 8

typedef struct
{
    void *data[QUEUE_SIZE];
    int head;
    int tail;
    int count;
    int futex_count; // futex для синхронизации
} BlockingQueue;
```

```c
static long futex_wait(int *uaddr, int val)
{
    return syscall(SYS_futex, uaddr, FUTEX_WAIT, val, NULL, NULL, 0);
}

static long futex_wake(int *uaddr, int val)
{
    return syscall(SYS_futex, uaddr, FUTEX_WAKE, val, NULL, NULL, 0);
}

BlockingQueue *queue_create()
{
    BlockingQueue *queue = (BlockingQueue *)malloc(sizeof(BlockingQueue));
    if (queue == NULL)
    {
        perror("malloc failed");
        exit(1);
    }
    queue->head = 0;
    queue->tail = 0;
    queue->count = 0;
    queue->futex_count = 0;
    return queue;
}

void enqueue(BlockingQueue *queue, void *data)
{
    int ret;
    while (queue->count == QUEUE_SIZE)
    {
        futex_wait(&queue->futex_count, queue->count);
    }
    queue->data[queue->tail] = data;
    queue->tail = (queue->tail + 1) % QUEUE_SIZE;
    queue->count++;
    queue->futex_count = queue->count; // Обновляем futex
    ret = futex_wake(&queue->futex_count, 1);
    if (ret == -1)
        perror("futex_wake failed");
}

void *dequeue(BlockingQueue *queue)
{
    int ret;
    void *data;
    while (queue->count == 0)
    {
        futex_wait(&queue->futex_count, queue->count);
    }
    data = queue->data[queue->head];
```

```
    queue->head = (queue->head + 1) % QUEUE_SIZE;
    queue->count--;
    queue->futex_count = queue->count; // Обновляем futex
    ret = futex_wake(&queue->futex_count, 1);
    if (ret == -1)
        perror("futex_wake failed");
    return data;
}

void queue_destroy(BlockingQueue *queue)
{
    free(queue);
}

////////////////////////////////////////////////////////////

#define NUM_OF_PROD 4
#define MAX_VOLUME_OF_STORAGE 50

// Структура Продукт
typedef struct Prod
{
    int num;
    char name[16];
} Prod;

// Структура производительность
// конкретной продукции
typedef struct Efficiency
{
    Prod prod;
    unsigned long long time;
    int amount;
} Efficiency;

// Структура данных, которые
// поступают в очередь, для
// изменения склада
typedef struct QueueData
{
    int num_obj;
    int prod_num;
    int add_val;
} QueueData;

int storage[NUM_OF_PROD]; // склад
BlockingQueue *queue;     // очередь
int futex_val;            // значение для передачи в futex

// sleep в миллисекундах
void sleep_ms(int milliseconds) { usleep(milliseconds * 1000); }
```

```
// Функция для определения наличия свободного пространства на складе
int get_free_storage_space()
{
    // requierd lock storage
    int sum = 0;
    for (size_t i = 0; i < NUM_OF_PROD; i++)
    {
        sum += storage[i];
        if (sum > MAX_VOLUME_OF_STORAGE)
        {
            return 0;
        }
    }

    return MAX_VOLUME_OF_STORAGE - sum;
}

// Поток фабрики (писатель),
// производит одну продукцию
void run_fabr(
    const int fabr_num,
    const Efficiency eff)
{
    printf("efficiency of fabric number %d\n", fabr_num);
    printf("\tname: %s, amount: %d, time: %llu\n",
           eff.prod.name, eff.amount, eff.time);

    while (1)
    {
        sleep_ms(eff.time);

        QueueData *queueData = (QueueData *)malloc(sizeof(QueueData));
        queueData->num_obj = fabr_num;
        queueData->prod_num = eff.prod.num;
        queueData->add_val = eff.amount;
        enqueue(queue, queueData);

        printf("fabr %d do %s in amount of %d\n",
               fabr_num, eff.prod.name, eff.amount);
    }
}

// Поток головы (писатель), случайно
// съедает единицу указанной продукции
void run_head(
    const int head_num,
    const Prod eating_prod[],
    const int amount_eating_prod,
    const unsigned long long eating_time)
{
```

```
    printf("head %d eat:\n\t", head_num);
    for (size_t i = 0; i < amount_eating_prod; i++)
        printf("%s, ", eating_prod[i].name);
    printf("\b\b - eating per %llu\n", eating_time);

    while (1)
    {
        srand(time(0));
        sleep_ms(eating_time);

        int index = rand() % amount_eating_prod;
        printf("eating prod - %s\n", eating_prod[index].name);

        QueueData *queueData = (QueueData *)malloc(sizeof(QueueData));
        queueData->num_obj = head_num;
        queueData->prod_num = eating_prod[index].num;
        queueData->add_val = -1;
        enqueue(queue, queueData);

    }
}

void run_storage_manager()
{
    while (1)
    {
        QueueData *queueData = dequeue(queue);

        printf("- input queue data: %d %d\n",
               queueData->prod_num, queueData->add_val);

        storage[queueData->prod_num] += queueData->add_val;
        if (storage[queueData->prod_num] < 0 || !get_free_storage_space())
        {
            storage[queueData->prod_num] -= queueData->add_val;
            if (queueData->add_val < 0)
                printf("head %d cant eat prodnum: %d\n", queueData->num_obj,
queueData->prod_num);
            else
                printf("fabr %d cant put prodnum: %d\n", queueData->num_obj,
queueData->prod_num);
        }
        else
        {
            if (queueData->add_val < 0)
                printf("1 head %d eat prodnum: %d\n", queueData->num_obj,
queueData->prod_num);
            else
                printf("1 fabr %d put prodnum: %d\n", queueData->num_obj,
queueData->prod_num);
        }
```

```
        printf("- storage state: ");
        for (size_t i = 0; i < NUM_OF_PROD; i++)
            printf("%d ", storage[i]);
        printf("\n");

        free(queueData);
    }
}

int main()
{
    queue = queue_create();

    Prod PROD_PIE = {0, "pie"};
    Prod PROD_CAKE = {1, "cake"};
    Prod PROD_CANDY = {2, "candy"};
    Prod PROD_BREAD = {3, "bread"};

    Efficiency eff1 = {
        PROD_PIE,
        50,
        2};
    Efficiency eff2 = {
        PROD_CAKE,
        70,
        1};
    Efficiency eff3 = {
        PROD_CANDY,
        40,
        3};
    Efficiency eff4 = {
        PROD_BREAD,
        60,
        1};

    pthread_t
        thr_storage_manager,
        thr_head1,
        thr_head2, thr_head3,
        thr_fabr11, thr_fabr12,
        thr_fabr21, thr_fabr22;

    void *fabr_1_1()
    {
        run_fabr(1, eff1);
        return 0;
    };
    pthread_create(&thr_fabr11, 0, fabr_1_1, 0);

    void *fabr_1_2()
```

```
    {
        run_fabr(1, eff2);
        return 0;
    };
    pthread_create(&thr_fabr12, 0, fabr_1_2, 0);

    void *fabr_2_1()
    {
        run_fabr(2, eff3);
        return 0;
    };
    pthread_create(&thr_fabr21, 0, fabr_2_1, 0);

    void *fabr_2_2()
    {
        run_fabr(2, eff4);
        return 0;
    };
    pthread_create(&thr_fabr22, 0, fabr_2_2, 0);

    void *storage_manager()
    {
        run_storage_manager();
        return 0;
    }
    pthread_create(&thr_storage_manager, 0, storage_manager, 0);

    void *head_1()
    {
        Prod eating_prod[] = {PROD_CAKE, PROD_CANDY};
        run_head(1, eating_prod, 2, 80);
        return 0;
    };
    pthread_create(&thr_head1, 0, head_1, 0);

    void *head_2()
    {
        Prod eating_prod[] = {PROD_PIE, PROD_BREAD};
        run_head(2, eating_prod, 2, 90);
        return 0;
    };
    pthread_create(&thr_head2, 0, head_2, 0);

    void *head_3()
    {
        Prod eating_prod[] = {PROD_PIE, PROD_CAKE, PROD_CANDY, PROD_BREAD};
        run_head(3, eating_prod, 4, 70);
        return 0;
    };
    pthread_create(&thr_head3, 0, head_3, 0);
```

```
    while (1) { }

    return 0;
}
```

Протоколы, логи, скриншоты, графики.

В программе есть основные функции для выполнения в потоке. Функция run_fabr получает информацию о продукции (внутри фабрики создание разной продукции происходит параллельно). Функция run_head получает список «поедаемой продукции», выбор элемента происходит случайно. Функция run_storage_manager считывает изменения на складе и или их принимает, или отклоняет в зависимости от свободного места.

Передача данных происходит через самописную блокирующую очередь, которая блокируется с помощью futex.

При производстве (или «поедании») не учитывается размерность склада, поэтому продукция зачастую «выкидывается» (или «не съедается»).

Вывод программы:

```
efficiency of fabric number 1
        name: pie, amount: 2, time: 50
efficiency of fabric number 1
        name: cake, amount: 1, time: 70
head 1 eat:
        cake, candy - eating per 80
head 2 eat:
        pie, bread - eating per 90
head 3 eat:
        pie, cake, candy, bread - eating per 70
efficiency of fabric number 2
        name: bread, amount: 1, time: 60
efficiency of fabric number 2
        name: candy, amount: 3, time: 40
fabr 2 do candy in amount of 3
- input queue data: 2 3
1 fabr 2 put prodnum: 2
- storage state: 0 0 3 0
fabr 1 do pie in amount of 2
- input queue data: 0 2
1 fabr 1 put prodnum: 0
- storage state: 2 0 3 0
- input queue data: 3 1
1 fabr 2 put prodnum: 3
- storage state: 2 0 3 1
fabr 2 do bread in amount of 1
eating prod - cake
```

```
- input queue data: 1 -1
0 head 3 cant eat prodnum: 1
- storage state: 2 0 3 1
fabr 1 do cake in amount of 1
- input queue data: 1 1
1 fabr 1 put prodnum: 1
- storage state: 2 1 3 1
eating prod - candy
- input queue data: 2 -1
1 head 1 eat prodnum: 2
- storage state: 2 1 2 1
```

Замеры. С помощью счётчика объявленного, как volatile был произведён замер съеденной продукции при 1, 3, 6 ядрах за 10 секунд. В каждом замере по 5 экспериментов

количество ядер	значения	среднее значение
1	322, 326, 353, 326, 331	331,6
3	323, 253, 363, 352, 350	328,2
6	257, 296, 329, 307, 292	296,2

Заметен интересный парадокс, что чем больше ядер, тем меньше эффективность, хотя предполагался обратный результат (особенно с 3мя ядрами, количество нередко было не больше 90). Возможно, что на результат повлияло наличие других процессов, которые в зависимости от свободных ядер совершали сервисные действия. Также выборка была довольно небольшая.

Задание 2.
Текст программы на языке C с комментариями.

```c
#define NUM_OF_PROD 4
#define MAX_VOLUME_OF_STORAGE 50
#define CHANNEL_NAME "fifo.txt"

// Структура Продукт
typedef struct Prod
{
    int num;
    char name[16];
} Prod;

// Структура производительность
// конкретной продукции
typedef struct Efficiency
{
    Prod prod;
    unsigned long long time;
```

```
    int amount;
} Efficiency;

// Структура данных, которые
// поступают в очередь, для
// изменения склада
typedef struct Data
{
    int num_obj;
    int prod_num;
    int add_val;
} Data;

int storage[NUM_OF_PROD]; // склад
sem_t sem;
int fd_fifo;

// sleep в миллисекундах
void sleep_ms(int milliseconds) { usleep(milliseconds * 1000); }

// Функция для определения наличия свободного пространства на складе
int get_free_storage_space()
{
    // requierd lock storage
    int sum = 0;
    for (size_t i = 0; i < NUM_OF_PROD; i++)
    {
        sum += storage[i];
        if (sum > MAX_VOLUME_OF_STORAGE)
        {
            return 0;
        }
    }

    return MAX_VOLUME_OF_STORAGE - sum;
}

// Поток фабрики (писатель),
// производит одну продукцию
void run_fabr(
    const int fabr_num,
    const Efficiency eff)
{
    printf("efficiency of fabric number %d\n", fabr_num);
    printf("\tname: %s, amount: %d, time: %llu\n",
           eff.prod.name, eff.amount, eff.time);

    while (1)
    {
        sleep_ms(eff.time);
```

```
        Data *data = (Data *)malloc(sizeof(Data));
        data->num_obj = fabr_num;
        data->prod_num = eff.prod.num;
        data->add_val = eff.amount;

        printf("fabr %d do %s in amount of %d\n",
               fabr_num, eff.prod.name, eff.amount);

        sem_wait(&sem);

        fd_fifo = open(CHANNEL_NAME, O_WRONLY | O_NONBLOCK);
        write(fd_fifo, data, sizeof(data));
        close(fd_fifo);

        sem_post(&sem);
    }
}

// Поток головы (писатель), случайно
// съедает единицу указанной продукции
void run_head(
    const int head_num,
    const Prod eating_prod[],
    const int amount_eating_prod,
    const unsigned long long eating_time)
{
    printf("head %d eat:\n\t", head_num);
    for (size_t i = 0; i < amount_eating_prod; i++)
        printf("%s, ", eating_prod[i].name);
    printf("\b\b - eating per %llu\n", eating_time);

    while (1)
    {
        sleep_ms(eating_time);

        srand(time(0));
        int index = rand() % amount_eating_prod;

        Data *data = (Data *)malloc(sizeof(Data));
        data->num_obj = head_num;
        data->prod_num = eating_prod[index].num;
        data->add_val = -1;

        printf("eating prod - %s\n", eating_prod[index].name);

        sem_wait(&sem);

        fd_fifo = open(CHANNEL_NAME, O_WRONLY | O_NONBLOCK);
        write(fd_fifo, data, sizeof(Data *));
        close(fd_fifo);
```

```
        sem_post(&sem);
    }
}

void run_storage_manager()
{
    fd_fifo = open(CHANNEL_NAME, O_RDONLY);

    while (1)
    {
        Data* data = 0;

        sem_wait(&sem);

        read(fd_fifo, data, sizeof(data));

        sem_post(&sem);

        if (data != 0)
        {
            printf("- input queue data: %d %d\n",
                   data->prod_num, data->add_val);

            storage[data->prod_num] += data->add_val;

            if (storage[data->prod_num] < 0 || !get_free_storage_space())
            {
                storage[data->prod_num] -= data->add_val;
                if (data->add_val < 0)
                    printf("0 head %d cant eat prodnum: %d\n", data->num_obj,
data->prod_num);
                else
                    printf("0 fabr %d cant put prodnum: %d\n", data->num_obj,
data->prod_num);
            }
            else
            {
                if (data->add_val < 0)
                    printf("1 head %d eat prodnum: %d\n", data->num_obj, data-
>prod_num);
                else
                    printf("1 fabr %d put prodnum: %d\n", data->num_obj, data-
>prod_num);
            }

            free(data);

            printf("- storage state: ");
            for (size_t i = 0; i < NUM_OF_PROD; i++)
                printf("%d ", storage[i]);
            printf("\n");
```

```
        }
        else
        {
            printf("stmg nothing in fifo\n");

            sleep(1);
        }
    }

    close(fd_fifo);
}

int main()
{
    Prod PROD_PIE = {0, "pie"};
    Prod PROD_CAKE = {1, "cake"};
    Prod PROD_CANDY = {2, "candy"};
    Prod PROD_BREAD = {3, "bread"};

    Efficiency eff1 = {
        PROD_PIE,
        50,
        2};
    Efficiency eff2 = {
        PROD_CAKE,
        70,
        1};
    Efficiency eff3 = {
        PROD_CANDY,
        40,
        3};
    Efficiency eff4 = {
        PROD_BREAD,
        60,
        1};

    unlink(CHANNEL_NAME);
    if ((mkfifo(CHANNEL_NAME, 0777)) == -1)
    {
        fprintf(stderr, "Невозможно создать fifo\n");
        exit(0);
    }

    sem_init(&sem, 0, 1);

    if (fork() == 0)
        run_fabr(1, eff1);
    if (fork() == 0)
        run_fabr(1, eff2);
    if (fork() == 0)
        run_fabr(2, eff3);
```

```
    if (fork() == 0)
        run_fabr(2, eff4);
    if (fork() == 0)
    {
        Prod eating_prod1[] = {P ROD_CAKE, PROD_CANDY};
        run_head(1, eating_prod1, 2, 80);
    }
    if (fork() == 0)
    {
        Prod eating_prod2[] = {PROD_PIE, PROD_BREAD};
        run_head(2, eating_prod2, 2, 90);
    }
    if (fork() == 0)
    {
        Prod eating_prod3[] = {PROD_PIE, PROD_CAKE, PROD_CANDY, PROD_BREAD};
        run_head(3, eating_prod3, 4, 70);
    }

    if (fork() == 0)
        run_storage_manager();

    while (1)
    {
    }

    return 0;
}
```

Протоколы, логи, скриншоты, графики.

К сожалению, не удалось выполнить передачу с помощью именованного канала, предположительно из-за особенностей его работы.

Выводы

В ходе лабораторной работы было выполнено индивидуальное задание. Изучены различия между процессами и потоками в ОС Linux (Ubuntu), а также освоены некоторые механизмы синхронизации и межпроцессорного взаимодействия для обеспечения корректной работы программ в многозадачной среде.