Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»

КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №5 ПО ДИСЦИПЛИНЕ:
ТИПЫ И СТРУКТУРЫ ДАННЫХ

Обработка очередей

Студент **Павлов Д. В.**

Группа **ИУ7-33Б**

Вариант 3

Название предприятия **НУК ИУ МГТУ им. Н. Э. Баумана**

Студент _____ **Павлов Д. В.**

Преподаватель _____ **Никульшина Т. А.**

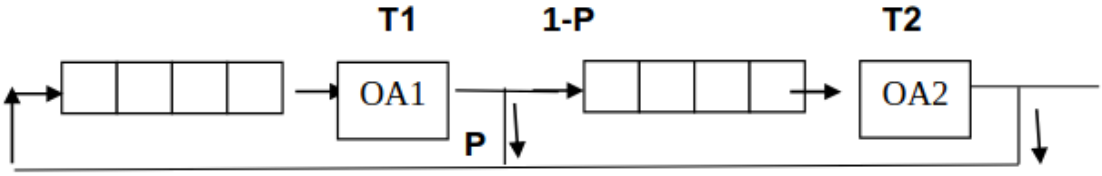
2024

1. Описание условия задачи.

Требуется смоделировать процесс обслуживания до выхода из ОА2 первых 1000 заявок. Выдавать на экран после обслуживания в ОА2 каждых 100 заявок информацию о текущей и средней длине каждой очереди, а в конце процесса общее время моделирования, время простоя ОА2, количество срабатываний ОА1, среднее времени пребывания заявок в очереди. Обеспечить по требованию пользователя выдачу на экран адресов элементов очереди при удалении и добавлении элементов. Проследить, возникает ли при этом фрагментация памяти.

2. Описание технического задания.

Система массового обслуживания состоит из двух обслуживающих аппаратов (ОА1 и ОА2) и двух очередей заявок. Всего в системе обращается 100 заявок.



Заявки поступают в "хвост" каждой очереди; в ОА они поступают из "головы" очереди по одной и обслуживаются по случайному закону за интервалы времени T1 и T2, равномерно распределенные от 0 до 6 и от 1 до 8 единиц времени соответственно. (Все времена – вещественного типа). Каждая заявка после ОА1 с вероятностью P=0.7 вновь поступает в "хвост" первой очереди, совершая новый цикл обслуживания, а с вероятностью 1-P входит во вторую очередь. В начале процесса все заявки находятся в первой очереди.

Входные данные.

Целое число: пункт меню (от 0 до 11).

```
1) Добавить элемент в очередь на массиве
2) Добавить элемент в очередь на списке
3) Извлечь элемент из очереди на массиве
4) Извлечь элемент из очереди на списке
5) Вывести текущее состояние очереди на массиве
6) Вывести текущее состояние очереди на списке
7) Вывести список свободных областей
8) Провести симуляцию процесса обслуживания очереди на массиве
9) Провести симуляцию процесса обслуживания очереди на списке
10) Изменить параметры симуляции
11) Вывести результат замеров удаления и добавления
0) Завершить работу программы
```

Целое число: номер элемента очереди.
Вещественные числа: интервалы времени обработки, вероятность.

Выходные данные.
Сообщение об успехе или ошибке выполнения пункта, текущее состояние очереди, результаты моделирования системы очередей, список освобождённых областей, таблица замеров по времени и памяти.

Способ обращения к программе.
Запускается через терминал командой *./app*

Возможные аварийные ситуации и ошибки пользователя.
Некорректный ввод данных, пустота или переполнение очереди. Ошибка выделения памяти.

3. Описание внутренних структур данных.

Структура заявки.
```
typedef struct
{
   int id; // Номер заявки
   double entered_time; // Время вхождения заявки в ОА
} request_t;
```

Структура для очереди на массиве.
```
#define MAX_ARR_SIZE 500

typedef struct
{
   request_t data[MAX_ARR_SIZE]; // Элементы очереди
   int size; // Размер очереди
   int max_size; // Максимальный размер очереди
   int head; // Индекс начала очереди
   int tail; // Индекс конца очереди
} arr_queue_t;
```

Узел списка.
```
typedef struct node
{
   request_t data;
   struct node *next;
```

```
} node_t;
```

Узел списка для освобожденных адресов.

```
typedef struct free_node
{
   void *address;
   struct free_node *next;
} free_node_t;
```

Структура для стека на списке.

```
typedef struct
{
   node_t *head; // Указатель на начало очереди
   node_t *tail; // Указатель на конец очереди
   int size; // Размер очереди
   int max_size; // Максимальный размер очереди

   free_node_t *free_list; // Список освобжденных элементов
   int free_elems_size; // Кол-во освобожденных элементов
} list_queue_t;
```

Структура для универсальной очереди

```
// Перечисление для выбора типа очереди
typedef enum {
   ARRAY_QUEUE,
   LIST_QUEUE
} queue_type_t;

// Универсальная структура очереди
typedef struct {
   queue_type_t type; // Тип очереди
   arr_queue_t arr_queue;  // Очередь на массиве
   list_queue_t list_queue;  // Очередь на списке
} queue_t;
```

Структура для обслуживающего аппарата.

```
typedef struct
{
   int is_busy; // Занят ли аппарат
   double end_time; // Время конца обслуживания
```

```
    request_t current_request; // Текущая заявка в аппарате
} service_unit_t;
```

Интерфейс для работы со структурами.

Для работы с очередью на массиве

```
// Инициализация очереди
void arr_queue_init(arr_queue_t *queue, int max_size);

// Проверка очереди на пустоту
int is_arr_queue_empty(arr_queue_t *queue);

// Проверка очереди на переполненность
int is_arr_queue_full(arr_queue_t *queue);

// Добавление элемента в очередь
void arr_enqueue(arr_queue_t *queue, request_t elem);

// Удаление элемента из очереди
request_t arr_dequeque(arr_queue_t *queue);

// Вывод элементов очереди на экран
void print_arr_queue(arr_queue_t *queue);
```

Для работы с очередью на списке

```
// Инициализация очереди
void list_queue_init(list_queue_t *queue, int max_size);

// Проверка очереди на пустоту
int is_list_queue_empty(list_queue_t *queue);

// Проверка очереди на переполненность
int is_list_queue_full(list_queue_t *queue);

// Добавление элемента в очередь
int list_enqueue(list_queue_t *queue, request_t elem);

// Удаление элемента из очереди
int list_dequeue(list_queue_t *queue, request_t *removed_elem);

// Вывод элементов очереди на экран
void print_list_queue(list_queue_t *queue);
```

```
// Очищение очереди
void free_list_queue(list_queue_t *queue);

// Вывод освобжденной области
void print_free_area(list_queue_t *queue);
```

Для работы с универсальной очередью

```
//Инициализация очереди
void queue_init(queue_t *queue, queue_type_t queue_type, int max_size);

//Проверка на пустоты очереди
int is_queue_empty(queue_t *queue);

//Проверка на заполненность очереди
int is_queue_full(queue_t *queue);

//Добавление элемента в очередь
int enqueue(queue_t *queue, request_t elem);

//Извлечение элememнта из очереди
int dequeue(queue_t *queue, request_t *removed_elem);

// Получение длины очереди
int queue_length(queue_t *queue);

//Очищение очереди
void queue_free(queue_t *queue);
```

Для моделирования очереди заявок

```
void simulation_queue(queue_t *queue1, queue_t *queue2, double t1_start, double t1_end, double t2_start, double t2_end, double probability);
```

4. Описание алгоритма.

Первая очередь наполняется 100 заявками.
Далее программа работает пока из второй очереди не выйдет 1000 заявок.

1. Работа первого и второго аппарата.
    - Проверяется не пуста ли очередь и свободен ли аппарат.
    - Если свободен, то извлекается элемент из первой очереди.
    - Аппарат помечается как занятый.

- Устанавливается текущая заявка для аппарата.
- Вычисляется время обслуживания, которое будет случайным образом выбрано из заданного диапазона временных значений, и устанавливается время завершения обработки заявки .

2. Работа второго аппарата
   - Проверяется не пуста ли очередь и свободен ли аппарат.
3. Определение ближайшего времени завершения
   - Если оба аппарата заняты, выбирается тот, чьё время завершения меньше.
   - Если только один аппарат занят, следующее время берётся от занятого аппарата.
4. Обновление времени и учёт простоя второго аппарата
   - Если второй аппарат (ОА2) свободен, вычисляется время простоя второго аппарата и добавляется к общему времени простоя
   - Устанавливается текущее время на время завершения обработки
5. Завершение обработки заявки
   - Если завершение обработки принадлежит первому аппарату (ОА1), то:
     - Решается, в какую очередь добавить завершённую заявку на основе случайного числа и вероятности: либо в первую очередь, либо во вторую.
     - Устанавливается статус аппарата обратно в свободный.
   - Если завершение обработки принадлежит второму аппарату (ОА2):
     - Завершённая заявка добавляется в queue1.
     - Второй аппарат помечается как свободный.
     - Увеличивается счётчик обработанных запросов.

5. Тестирование задания.

Так как по заданию два аппарата и все заявки изначально лежат в первой очереди, то теоретический расчёт общего времени работы системы будет определяться временем обработки в наиболее загруженном аппарате.

Соответственно время работы первого аппарата = среднее время обработки заявки * на количество обработанных заявок, и второго = среднее время обработки заявки * на количество обработанных заявок в этом аппарате.

Интервал обработки первого аппарата = [0, 6], второго — [1, 8], с вероятностью возврата в первую очередь = 0.7, и выходом из второго аппарата 1000 заявок. Тогда время обработки первого аппарата = 3 * (1000 * (1 / (1 — 0.7)) = 10000 Второго аппарата = 1000 * 3.5 = 3500.

Тогда общее время моделирования будет равно 10000.

Реальное время работы при таких данных:

На очереди на массиве:

```
Итоги симуляции:
Общее время работы: 10217.18
Теоретический рассчёт: 10167.00
Процент погрешности: 0.49%
Время простоя ОА2: 5779.77
Количество срабатываний ОА1: 3389
Средняя длина первой очереди 97.90
Средняя длина второй очереди: 1.10
```

На очереди на списке:

```
Итоги симуляции:
Общее время работы: 10300.17
Теоретический рассчёт: 10000.00
Процент погрешности: 3.00%
Время простоя ОА2: 5834.91
Количество срабатываний ОА1: 3461
Средняя длина первой очереди 98.40
Средняя длина второй очереди: 0.60
```

Можно заметить, что погрешность не превышает 3%, что удовлетворяет поставленному условию.

Проверка на фрагментацию:

```
Очередь после удаления 5 элементов:
Адресс элемента - 0x5b9df74f5c80, значение элемента - 5
Адресс элемента - 0x5b9df74f5ca0, значение элемента - 6
Адресс элемента - 0x5b9df74f5cc0, значение элемента - 7
Адресс элемента - 0x5b9df74f1f00, значение элемента - 8
Адресс элемента - 0x5b9df74f1ee0, значение элемента - 9
Список освобожденных областей после удаления:
Список свободных областей:
Освобожденная область памяти: 0x5b9df74f5c60
Освобожденная область памяти: 0x5b9df74f5c40
Освобожденная область памяти: 0x5b9df74f5c20
Освобожденная область памяти: 0x5b9df74f5c00
Освобожденная область памяти: 0x5b9df74f5be0
```

```
Очередь после добавления 5 новых элементов:
Адресс элемента - 0x5b9df74f5c80, значение элемента - 5
Адресс элемента - 0x5b9df74f5ca0, значение элемента - 6
Адресс элемента - 0x5b9df74f5cc0, значение элемента - 7
Адресс элемента - 0x5b9df74f1f00, значение элемента - 8
Адресс элемента - 0x5b9df74f1ee0, значение элемента - 9
Адресс элемента - 0x5b9df74f5c60, значение элемента - 10
Адресс элемента - 0x5b9df74f1ea0, значение элемента - 11
Адресс элемента - 0x5b9df74f1e80, значение элемента - 12
Адресс элемента - 0x5b9df74f1e60, значение элемента - 13
Адресс элемента - 0x5b9df74f1e20, значение элемента - 14
```

Можно заметить, что память не чередуется (выделяется последовательно), и можно предположить, что фрагментации не возникает.

6. Замеры.

Количество итераций = 10000.

Временные замеры

Добавление элементов

Количество элементов	Время для очереди на массиве	Время для очереди на списке	Преимущество выполнения стека на массиве к стеку на списке (%)
10	0.22	0.69	68.11
50	0.78	2.87	73
100	1.36	5.47	75
335	4.70	23.05	80
500	7.00	31.16	78

Удаление элементов

Количество элементов	Время для стека на массиве	Время для стека на списке	Преимущество выполнения стека на массиве к стеку на списке (%)
10	0.22	0.48	54
50	0.72	1.80	59
100	1.50	5.48	73
130	4.71	12.51	63
500	7.40	19.93	63

Замеры по памяти

Длина последовательности	Память для стека на массиве (байт)	Память для стека на списке (байт)	Преимущество по памяти стека на массиве к стеку на списке (%)
10	8000	240	96.99
50	8000	1200	83.33
100	8000	2400	71.43
335	8000	8040	0.49
500	8000	12000	33.33

7. Выводы по проделанной работе.

Очередь на массиве оказывается более эффективным по времени. Её фиксированный размер и доступ по индексу обеспечивают высокую скорость операций, что позволяет обрабатывать данные быстрее, чем в очереди на списке, где работа с указателями создаёт дополнительную нагрузку. По памяти можно заметить, что при заполненности массива до 70%, очередь на списке будет выигрывать, а такое заполнение обычно встречается в реальных задачах, вследствие чего можно сделать, что список памяти обычно выигрывает по памяти. На основе замеров 4 лабораторной работы, также можно сделать вывод, что реализация LIFO работает на 20-30% быстрее чем реализация FIFO.

Контрольные вопросы.

1. Что такое FIFO и LIFO?

FIFO и LIFO — это принципы функционирования абстрактных типов данных таких как очередь и стек.
FIFO (First In First Out) переводится как первый пришел, первый вышел, то есть процессы протекают как в реальной очереди. Данные обрабатываются в том порядке, в котором они поступают. Элементы добавляются в конец структуры данных, а извлекаются из начала.
LIFO (Last In First Out) переводится как последний пришел, первый вышел, чт сопоставимо со стпокой книг. Данные обрабатываются в обратном порядке их поступления. Элементы добавляются и извлекаются с одного конца структуры данных.

2. Каким образом, и какой объем памяти выделяется под хранение очереди при различной ее реализации?

При реализации очереди с помощью списка каждый узел требует динамического выделения памяти, что приводит к большему объему занимаемой памяти из-за дополнительных указателей (4 байта на 32-битной и 8 байт на 64-битной архитектуре).
При использовании статического массива память выделяется заранее и занимает фиксированное количество байт в зависимости от типа данных, что обеспечивает меньшие накладные расходы и более эффективное использование памяти.
3. Каким образом освобождается память при удалении элемента из очереди при ее различной реализации?

При хранении очереди связанным списком, верхний элемент удаляется путём освобождения памяти для него и смещения указателя, указывающего на начало стека. При удалении из очереди, реализованной массивом, смещается лишь

указатель на голову очереди, а сами данные остаются в памяти, но считаются «стёртыми».

4. Что происходит с элементами очереди при ее просмотре?

При просмотре очереди её элементы извлекаются из него, причем классическая реализация очереди предполагает, что просмотреть содержимое очереди без извлечения (удаления) её элементов невозможно.

5. От чего зависит эффективность физической реализации очереди?

Эффективность реализации очереди массивом или списком зависит от ресурсов памяти и начального размера очереди. Если нам важна занимаемая очередью память и скорость обработки, то эффективнее использовать очередь на массиве, а если мы не знаем сколько элементов может храниться в очереди, то лучше использовать очередь на списке.

6. Каковы достоинства и недостатки различных реализаций очереди в зависимости от выполняемых над ней операций?

Очередь на массиве, обычно более тяжело реализуема, нежели чем на списке, но она обычно выигрывает по скорости. Она эффективно использует память при фиксированном размере очереди, но требует заранее знать её максимальный размер или динамически перераспределять память, что может усложнять работу. Без кольцевого буфера извлечение элементов из начала требует сдвига, что замедляет операции. Кольцевой буфер очень эффективен для фиксированных объёмов данных, так как исключает сдвиг данных и минимизирует накладные расходы на память, но его реализация сложнее, а размер ограничен и требует контроля переполнения и пустоты.
Очередь на списке легко реализуема, она поддерживает динамическое изменение размера, что удобно, если длина очереди заранее неизвестна. Она обеспечивает быстрое добавление и извлечение элементов, так как не требует сдвига данных, но требует больше памяти для хранения дополнительных указателей и может быть медленнее из-за раздельного размещения данных в памяти.

7. Что такое фрагментация памяти, и в какой части ОП она возникает?

Фрагментация памяти — это явление, при котором в оперативной памяти (ОП) остаются небольшие свободные участки, которые не могут быть использованы для размещения новых данных, даже если общий объем свободной памяти достаточен. Она возникает на куче(heap).

8. Для чего нужен алгоритм «близнецов».

Алгоритм «близнецов» используется для динамического распределения памяти, при котором память поделена на блоки размером 2^n и при запросе выделяется блок, размер которого максимально приближен к требуемой памяти и для минимизации фрагментации блоков памяти.

9. Какие дисциплины выделения памяти вы знаете?

First-Fit(первый подходящий). Этот метод ищет первый блок свободной памяти, который достаточно велик, чтобы удовлетворить запрос на выделение памяти. Best-Fit(лучший подходящий). Этот метод ищет наилучший подходящий блок памяти, который будет наиболее близок по размеру к запрашиваемому объёму. Также существует Worst-fit(худший подходящий), память выделяется в самом большом свободном блоке

10. На что необходимо обратить внимание при тестировании программы?

Нужно обратить внимание на корректное выделение и освобождение памяти, а также отслеживать переполнение или пустоту очереди.

11. Каким образом физически выделяется и освобождается память при динамических запросах?

Когда программа запрашивает память, операционная система ищет подходящий свободный блок и регистрирует его в таблице занятых областей памяти. При освобождении памяти, операционная система удаляет информацию о этом блоке из этой таблицы