МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. М.В. Ломоносова
Факультет вычислительной математики и кибернетики

ОТЧЁТ К ЗАДАНИЮ
ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ ИГРЫ «ЖИЗНЬ»

студента 205 учебной группы факультета ВМК МГУ
Бугаевского Владимира Михайловича

Преподаватель: Герасимов Сергей Валерьевич

ВАРИАНТ: 2b

Москва, 2014

1 Постановка задачи

Необходимо реализовать сервер математической игры «Жизнь», позволяющей моделировать развитие популяции живых организмов, эволюционирующих по заданным генетическим законам, на примере клеточной поверхности.

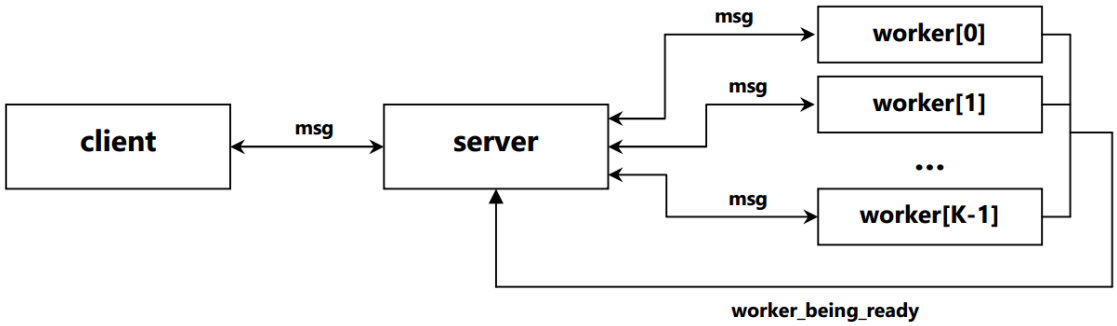
Процесс моделирования должен выполняться параллельными процессами-рабочими, каждый из которых отвечает за свой участок поверхности. Процесс-сервер получает команды, управляющие процессом моделирования, от процесса-клиента и перенаправляет их процессам-рабочим.

В качестве базового механизма обмена данными между процессами-рабочими должна быть использована разделяемая память.

В данном варианте задачи считается, что

1. клеточная поверхность является безграничной (ее верхняя граница соединена с нижней, левая - с правой), имеет форму тора;
2. поверхность разбивается вертикальные полосы, количество которых равно K.

2 Организация обмена данными между процессами



Все процессы связаны аппаратом очереди сообщений IPC через файл `server`. Существует особый «канал» между всеми рабочими и сервером `worker_being_ready` (назначение этого «канала» будет описано позже).

3 Особенности организации работы процесса-клиента

3.1 Инициализация процесса-клиента

На стадии инициализации клиент создает файл `server`, отвечающий за обмен сообщениями между процессами.

3.2 Проверка корректности вводимых данных

Наибольший интерес представляет проверка корректности разбиения «вселенной». По условию разбиение должно быть равномерным: ширина всех вертикальных полос должна быть одинаковой за возможным исключением только крайне правых областей. Такое разбиение не всегда возможно, например, в случае `N == 5 && K == 4` или `N == 7 && K == 5`. В таком случае, клиент предложит новое количество рабочих[1].

[1] `N` - число клеток «вселенной» по горизонтали; `K` - число процессов-рабочих.

3.3 Чтение команд из файла

Для упрощения работы, было принято решение предусмотреть возможность чтения потока команд из файла. В силу того, что в таком случае отправка команд серверу будет почти мгновенной, было решено расширить функционал клиента: добавить команду `sleep t`, которая будет приостанавливать чтение команд из файла на t секунд.

3.4 Удаление клетки

В качестве расширения функционала программы было решено добавить команду удаления клетки `del x y`. Эта команда чрезвычайно полезна, когда ввод осуществляется пользователем: благодаря ей он может исправить свою ошибку.

3.5 Завершение работы процесса-клиента

По окончании свой работы сервер удаляет файл `server`, отвечающий за обмен сообщениями между процессами.

4 Особенности организации работы процесса-сервера

4.1 Инициализация процесса-сервера

На стадии инициализации сервер создает два массива:

1. массив идентификаторов процессов `pid_t pid_worker[K]`;
2. массив распределения процессов `int pid_worker_map[N]`.

Сервер создает два файла: `worker-left` (левые границы областей вселенной) и `worker-right` (правые границы областей вселенной), а затем - семафоры и разделяемую память, связанные с каждой из границ областей вселенной.

После создания K процессов-рабочих сервер отправляет каждому из них информационное сообщение, содержащее размеры участка области «вселенной», соответствующего данному рабочему, и его номер. Целью данного сообщения, помимо передачи жизненно важной информации, - уведомить рабочего, что он может подключиться к разделяемой памяти, отвечающей за границы областей «вселенной».

4.2 Операция добавления (удаления) клетки

Для добавления (удаления) ячейки используется карта `pid_worker_map` распределения столбцов «вселенной». Она отвечает за распараллеливание процессов: в `pid_worker_map[i]` хранится номер рабочего, отвечающего за $i + 1$ столбец «вселенной» (столбцы нумеруются с 1).

```
int i = pid_worker_map[y-1];        /* за столбец y отвечает рабочий с номером i */
```

Т.о., существенно упрощается поиск соответствующего рабочего для добавления клетки на поле.

4.3 Синхронизация между процессом-сервером и процессами-рабочими

Сообщение между сервером и рабочими осуществляется следующим образом:

1. сервер посылает команду рабочим(ему);
2. сервер ожидает от них (него) уведомления о завершении работы.

За ожидание уведомления от рабочего отвечает функция `ssize_t server_waiting_worker(void)`, за отправку уведомления рабочим - функция `int worker_is_ready(void)`. Для того, чтобы сообщения приходящие от клиента не смешивались с уведомлениями от рабочих, было решено ввести новый тип сообщений `worker_being_ready`.

Замечание: Исключением данного правила является завершение процессов-рабочих. Сервер посылает команду `quit` и ждет завершение процессов средствами функции `wait()`

4.4 Организация условий атомарности для процессов-рабочих

Возникает проблема корректного выполнения команд `add`, `snapshot` и `stop` во время моделирования рабочими «вселенной». В качестве решения этой проблемы было выбрано поэтапное построение поколений. Сервер хранит счетчик оставшихся поколений `steps`. Пусть, `msg` - индикатор получения сервером команды от клиента.

1. Если `steps == 0`, то сервер находится в режиме ожидания команды от клиента.
2. Если `steps != 0 && msg != 0`, то он посылает рабочим команду построения следующего поколения.
3. Если `steps != 0 && msg != 0`, то он выполняет команду клиента.

4.5 Некорректный запуск процессов

Для решения проблемы аварийного завершения процессов используется аппарат сигналов: некорректно запустившийся процесс отправляет сигнал `SIGTERM` своему «родителю». В каждом из процессов «родителей» выставляется обработчик сигналов `void handler(int signo)`, который впоследствии запускает соответствующую функцию завершения процесса.

4.6 Завершение работы процесса-сервера

По окончании свой работы сервер удаляет все разделяемые ресурсы, созданные для рабочих.

5 Особенности организации процесса-рабочего

5.1 Инициализация процесса-рабочего

На стадии инициализации рабочий I создает массив указателей `char shmad[4]` для разделяемой памяти и массив семафоров `int semid[4]` для организации работы с этой памятью[2]:

- `shmad[0]` - правая граница левого соседа рабочего I;
- `shmad[1]` - левая граница рабочего I;
- `shmad[2]` - правая граница рабочего I;
- `shmad[3]` - левая граница правого соседа рабочего I.

На стадии инициализация разделяемая память заполняется `'.'`, в клетках нет жизни.

[2]Все разделяемые ресурсы уже созданы сервером.

5.2 Структура области «вселенной» процесса-рабочего

Рабочий в своем адресном пространстве хранит две карты: (1) карта моделируемого поколения и (2) карта предыдущего поколения - причем карты учитывают границы. По карте (2) рабочий строит карту (1). Рассмотрим пример подготовки карты (2):

- $M = 4$ - число клеток по вертикали в соответствующей области «вселенной» рабочего I;
- $N = 3$ - число клеток по горизонтали в области «вселенной» рабочего I.

Тогда карта (2) для этого рабочего будет выглядеть следующим образом:

X \ Y	0	1	2	3	4
0	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
4	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)
5	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)

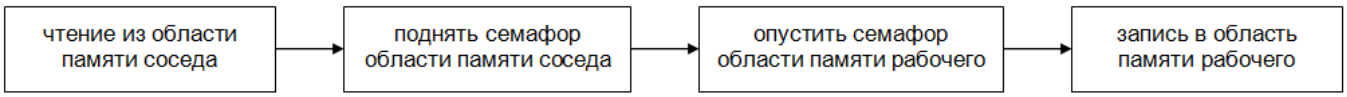
Зеленым цветом обозначена область «вселенной» рабочего I, **красным** - граничные области соседей рабочего I, **желтым** - области, получаемые копированием клеток карты.

Замечание: Такая карта удобна в частности тем, что позволяет легко посчитать количество соседей у граничных клеток.

5.3 Синхронизация процессов-рабочих при работе с разделяемой памятью

При работе с граничными областями частей «вселенной» необходима синхронизация, т.к. процесс-рабочий должен «успеть» получить данные из области границ соседа до того момента, как последний успеет их поменять.

В качестве такого средства организации синхронизации были выбраны семафоры IPC. Опишем более подробно работу с ними. Изначально все семафоры находятся в положении 0.



Пример:

1. читаем данные из `shmad[0]` и `shmad[3]` — `semid[4]={0,0,0,0}`;
2. поднимаем семафоры `semid[0]` и `semid[3]` — `semid[4]={1,0,0,1}`;
3. ждем пока рабочему разрешат писать — `semid[4]={0,1,1,1}`;
4. опускаем семафоры `semid[1]` и `semid[2]` — `semid[4]={0,0,0,1}`;
5. записать данные в `shmad[1]` и `shmad[2]` — `semid[4]={0,0,0,0}`.

Легко видеть, что при такой организации, сосед не может изменить свои данные пока их не получит рабочий[3].

Замечание: Данная схема применяется только для моделирования очередного поколения рабочими. Такой вид синхронизации не требуется, например, для команд `add` и `clear`. Это связано с атомарностью выполнения команд, посылаемых сервером (было описано выше).

5.4 Создание скриншота карты «вселенной»

Удобно воспользоваться пакетной пересылкой данных. Клиент отправляет пакет следующего вида с информацией со специальной мета-информацией:

```
message.mtype = pid_server;                          /* сообщение серверу          */
message.op    = O_SNAP;                              /* операция "snapshot"        */
message.prm1  = id_worker;                           /* индекс рабочего            */
message.prm2  = N;                                   /* длина строки своей области  *
                                                      * "вселенной"                */
memcpy(message.mtext, &map_state_curr[i][1], N);     /* копирует строку из своей    *
                                                      * области "вселенной"        */
message.mtext[N] = '\0';                             /* добавляем '\0' в качестве   *
                                                      * индикатора окончания строки */
```

Сервер собирает сообщение для клиента следующим образом:

```
rcv_worker_message(0);                               /* получить пакет от рабочего  */
int offset = message.prm1 * width;                   /* определяем смещение         *
                                                      * относительно начала строки  *
                                                      * для сервера                */
int len = message.prm2;                              /* сколько байт нужно вычитать */
memcpy(&msg[offset], message.mtext, len);            /* формирование строки         */
msg[N] = '\0';                                       /* индексация окончания строки */
```

Тогда клиенту остается лишь распечатать полученную строку.

[3]В указанном примере соседи рабочего могли сами опускать свои семафоры, если у них была такая возможность.

6 Предотвращение переполнения очереди сообщений

	1	2	3	4
1	ON	OFF	OFF	OFF
2	ON	ON	OFF	OFF
3	ON	OFF	ON	OFF
4	OFF	OFF	OFF	OFF

Выше приведен пример работы четырех процессов-рабочих с разделяемой памятью при построении очередного поколения. Граница отмечена **красным**, если в нее нельзя ничего записать, **желтым** — если уже можно в нее писать, **зеленым** — если она изменена[4]. **ON** соответствует тому, что рабочий находится в режиме ожидания, чтобы записать данные в свою границу, **OFF** — рабочий завершил свою работу.

Важно заметить, что сразу после завершения рабочего сервер должен принять от него соответствующее уведомление, в противном случае очередь сообщений рискует быть забитой этими уведомлениями (при больших K), и сервер заблокируется — мы получим *deadlock*. Поэтому для сервера отправка сообщений рабочим и прием от последних уведомлений будет иметь следующий вид:

```
int counter = 0;                                    /* счетчик показывает, сколько  *
                                                     * уведомлений от рабочих было  *
                                                     * принято                      */
for (int i = 0; i < K; i++) {
    while (snd_worker_message(i, 1) == -1) {        /* отправляем i-му рабочему     *
                                                     * сообщение, пока оно не уйдет */
        while (server_waiting_worker(1) != -1)      /* очередь сообщений забита:    *
            counter++;                               * сервер освобождает ее       */
    }
}
while (counter++ < K) server_waiting_worker(0);     /* дочитываем уведомления       */
```

[4]Рабочий может записать данные в правую границу только в том случае, если он уже что-то записал в левую.