Заметка разработчика

Документацией это сложно назвать, поэтому пока оставлю вышеупомянутое название. В этом документе я постараюсь донести до читателя по какому принципу работает симулятор виртуальной памяти и его логические части.

Для работы с симулятором в проекте должны быть следующие файлы:

* sim.cpp, sim.hpp – исходные файлы симулятора;
* vm.cpp, vm.hpp – исходные файлы с моделью системы, работающей с использованием принципов виртуальной памяти;
* main.cpp – файл, в котором указываются и запускаются агенты, устанавливается лимит на работу симулятора и осуществляется его запуск.

В этом симуляторе моделируются адреса, но не данные, которые хранятся по этим адресам. Структуризация памяти страничная, размер страницы указывается в байтах в виде степеней двойки (например: 2048 байт будет записано как 11, так как 211 = 2048).

В модели присутствуют следующие агенты: центральный процессор (ЦП), операционная система (ОС), архивная среда (АС) и процесс. Пару слов о каждом:

* центральный процессор (ЦП) выполняет только одну задачу – задачу преобразования адреса;
* операционная система (ОС) - многозадачный агент, в котором моделируется сама работа с виртуальной памятью, а также такие сущности, как таблицы переадресации (ТП), реальное адресное пространство (РАП), обработчик очереди претендентов на ЦП, планировщик ввода/вывода в/из АС;
* архивная среда (АС) занимается непосредственно моделированием ввода/вывода в АС, кстати архивное адресное пространство (ААП) также смоделировано в этом агенте;
* процесс – агент, в котором с смоделировано обращение к случайным адресам своего виртуального адресного пространства (ВАП), а также счётчик времени, которое осталось от кванта, на который процессу даются вычислительные мощности ЦП. Процессов может быть несколько, что нельзя сказать об остальных агентах.

Теперь опишу типичный сценарий работы этого симулятора по пунктам:

1. В **main.cpp** задаётся множество процессов, которое далее будет функционировать во всей системе, п ри этом у каждого объекта класса-агента процесс вызывается специальный метод **Start()**;
2. В методе **Start()** происходит планирование метода **LoadProcess()** класса-агента ОС, таким образом каждый процесс как бы “оповещает” ОС о своем существовании;
3. Узнав об очередном новом процессе, ОС выделяет память под его ТП и добавляет этот процесс в очередь претендентов на ЦП.
4. Как только хоть один процесс появляется в очереди, ОС немедленно начинает её обработку вызовом метода **ProcessQueue()**, процесс начинает работать в своем ВАП. Эта работа продолжается непродолжительный промежуток времени, называемый квантом, когда очередной процесс “исчерпал” свой квант времени, обработчик ставит этот процесс в конец очереди, а сам предоставляет ЦП другому претенденту, это смоделировано в методе **ChangeQueue().**

Работа процесса всегда происходит по такому алгоритму:

1. Определить, к какому адресу из ВАП будет обращаться процесс.
2. Спланировать метод **Convert()** у класса-агента ЦП.
3. Если квант времени процесса еще не исчерпан, то отнять от счётчика времени в очередном процессе время, которое занимает метод **Convert()**, и заново спланировать работу этого процесса, т.е. переход к п. 1, в ином случае спланировать смену очереди – метод **ChangeQueue()**.

Ниже немного подробностей про то, как и по какому принципу происходит **проверка на “исчерпание” кванта времени**.

Прежде всего стоит сказать, что счетчик кванта времени инкапсулирован в каждом отдельном процессе (хотя, вообще говоря, можно было бы вести этот подсчет и в классе ОС, но это на самом деле не суть важно). Ну и повторюсь, что сама проверка выполняется следующим образом:

Если счетчик времени > 0, значит квант времени еще не исчерпан, отнимаем от счётчика необходимое количество времени (например, время, необходимое на преобразование адреса – метод **Convert()**) и, соответственно, планируем следующий метод **Work()**, в ином случае планируем метод **ChangeQueue()**.

Несложно отметить, что при любых обстоятельствах рано или поздно в очередь претендентов добавляется претендент в силу того, что в симуляторе должны присутствовать объекты всех классов-агентов, а процесс – хотя бы в единичном экземпляре.

Очередь претендентов обрабатывается следующим образом:

1. Если на момент добавления претендента очередь пуста, планируем событие обработки очереди **ProcessQueue()**.
2. Событие **ProcessQueue()** устанавливает для очередного процесса квант времени (устанавливается счетчик времени), планируется работа этого процесса.

Если подвести итоги вышесказанному, можно выделить следующую цепочку событий: Process.Start() -> OS.LoadProcess() -> OS.Scheduler.AddProcess() -> OS.ProcessQueue() -> Process.Work() -> CPU.Convert() -> в случае, если квант не исчерпан еще раз Process.Work() -> …, в случае если исчерпан, то OS.ChangeQueue() -> OS.ProcessQueue() -> …

Другими словами, каждое исполнение метода Work() порождает одно исполнение метода Convert(), которое, в свою очередь, через проверку на исчерпание кванта породит еще один метод Work() или же, в случае прерывания, через его обработку, методы Allocate() и Substitute() также через проверку на исчерпание кванта породит еще один метод Work(). Круг замкнут, именно поэтому модель непрерывно работает.

**Замечание:** хоть и метод Convert() находится в классе-агенте ЦП, на деле же получается, что в этом методе происходит поиск нужной ТП, а затем и нужной записи этой ТП. Этот поиск реализован в методе **FindTT** у операционной системы (ОС), а значит ЦП только вызывает этот метод.

Пожалуй, из всего сценария остался только один не упомянутый момент. А что, если преобразование адреса не произойдет успешно, то есть в найденной записи ТП окажется, что реальный адрес не действителен? Специально для этого разработан метод HandelInterruption() у класса-агента ОС, который как раз моделирует обработку прерывания по отсутствию страницы. Итак, вот как это происходит:

1. \*происходит прерывание по отсутствию страницы\*, планируется метод **HandelInterruption().**
2. Если виртуальный адрес из-за преобразования которого произошло прерывание по отсутствию страницы был ранее загружен в АС, необходимо запланировать запрос на выгрузку данных из АС.
3. Вызываем метод **Allocate()** класса-агента ОС, который попытается найти в РАП свободный (ранее нераспределенный ни разу) реальный адрес. Если такой адрес найден, можно сказать, что обработка прерывания завершена, только перед тем, как планировать последующую работу процесса, также как и в методе **Convert(),** проверим, не исчерпан ли квант; если же свободных адресов нет, вызываем метод **Substitute()**.
4. В методе **Substitute()** класса-агента ОС моделируется задача поиска претендента на перераспределение адреса, **в текущей модели этот алгоритм основан на принципе случайного выбора**, когда претендент будет найден, спланируем работу процесса, в очередной раз проверив, чтобы квант времени не был исчерпан.

Последнее, что хотелось бы отметить, так это то, каким образом обрабатывается очередь запросов к АС. Главная особенность модели заключается в том, что очередь может обрабатываться (и так чаще всего и происходит) параллельно другим действия системы. Например, ЦП уже вызывает прерывание по отсутствию страницы для некоторого процесса под именем Process10, а в это же время АС выполняет загрузку данных по виртуальному адресу процесса Process09.

Обработка очереди запросов к АС происходит, в целом, по тому же принципу, что и обработка очереди претендентов, за тем лишь одним исключением, что ввод/вывод данных в/из АС происходит только тогда, когда в очереди есть хоть один запрос.