Дизайн-документ
Планировщик SCHED_RR.

Для реализации планировщика был создан список всех активных(ENV_RUNNING или ENV_RUNNABLE) процессов. Добавить процесс в этот список и удалить оттуда можно с помощью функций add_env и extract_env. Так как реализуется планировщик с приоритетами, то для каждого приоритета есть свой список. При вызове планировщика, если текущая среда не пуста, она помещается в конец соответствующего ей списка(каждой структуре Env было добавлено поле priority). Далее из первого непустого списка активных процессов(с высшим приоритетом) берется процесс и становится текущим. В случае, когда активных процессов нет, но есть ожидающие процессы с ограничением по времени, пробуждается наиболее приоритетный такой процесс(будет полезно в экстремальных ситуациях при реализации мьютексов). Для управления приоритетом процессов был реализован системный вызов sched_param. Он перемещает заданный процесс из списка с одним приоритетом в соответствующий список с другим(заявленным), а далее, если его приоритет стал больше приоритета текущего процесса или приоритет текущего процесса понизился, вызывается функция передачи управления(так как, по стандарту, как только более приоритетный процесс готов к исполнению, он вытесняет текущий). Также при системном вызове set_status также может произойти вытеснение текущего процесса. При очистке среды процесс удаляется из списка активных. Также каждому процессу выделен некоторый квант времени на выполнение. Контроль за этим квантом выполняется на основе использования RTC (которые генерируют периодические прерывания) и TSC (когда прерывание приходит или текущий процесс сменяется, мы считываем значение TSC, преобразуем его во время в миллисекундах с помощью функции timer_stop и вычитаем из текущего значения кванта. Если оно стало меньше нуля, то ставим новый процесс на выполнение). TSC удобен тем, что делает управление более гибким. В самом деле, если, например, контекст сменился и в этот же момент пришло прерывание по таймеру, то будет несправедливо отнимать период прерывания из кванта только что поставленного на выполнение процесса. Уменьшение кванта происходит в специальной функции decrease. Обновление кванта происходит в тот момент, когда процесс ставится на выполнение. Приоритет процесса наследуется его потомками. Для удовлетворения стандарту POSIX число приоритетов равно 32, а также было создано несколько функций, позволяющих узнать максимальный и минимальный уровни приоритета и квант времени.

Мьютексы

При реализации мьютексов пришлось столкнуться с несколькими проблемами:
Где реализовывать мьютексы? В пространстве пользователя или ядра? С одной стороны, если реализовывать их в ядре, то придется смириться с накладными расходами на системные вызовы(каждая операция обращения к мьютексу будет превращена в системный вызов). Также не удастся вернуть пользователю некоторый адрес в ядре, по которому мьютекс будет выделен. И, наконец, нужно будет продумать механику информационной безопасности, чтобы пользователь с их помощью не смог подвергнуть пространство ядра угрозе. Но у такого подхода есть и преимущества: во-первых, простота реализации разделяемых мьютексов. Если возвращать не сам мьютекс, а его идентификационный номер, то любой процесс сможет к нему обратиться(если реализовывать разделяемые мьютексы в юзерспейсе, то, скорее всего, их придется помещать в разделяемую память, а это лишние накладные расходы и реализация заметно усложнится). Во-вторых, внутри мьютексов не нужны будут вспомогательные системные вызовы(например, чтобы пробудить один из спящих потоков). В-третьих, мьютексы используют внутри себя спинлоки для контроля за критическими секциями внутри них самих(например, для установки нового приоритета владельца или выхода из мьютекса). Спинлоки же реализованы в JOS в пространстве ядра. Таким образом, для некоторого упрощения реализации мьютексов было решено сделать их частью ядра. Функция mutex_create возвращает id выделенного мьютекса или -1, если лимит мьютексов исчерпан. Функция mutex_lock принимает три параметра:id мьютекса, try-флаг(если он равен одному, то будет активирована семантика try-lock) и время ожидания. Если оно неположительно, это будет расценено как бесконечность. Если активирована try-семантика, при неудаче захвата эта функция сразу вернется. Иначе процесс будет помещен в очередь ждущих процессов, приоритет процесса-владельца мьютекса поднимется до приоритета процесса, вставшего в очередь, при необходимости. После того, как встать в очередь, процесс вызовет sched_yield и перейдет в режим ENV_WAITING или ENV_WAITING_WITH_TIME. Выйти из него он может по нескольким причинам: во-первых, он будет определен как следующий владелец мьютекса. Во-вторых, его квант времени ожидания исчерпается. И, в-третьих, мьютекс, ассоциированный с процессом, будет уничтожен. В первом случае он получит контроль над мьютексом, когда продолжит свое выполнение. Во втором-он понизит приоритет процесса-владельца до исходного, если их приоритеты совпадают, а далее приоритет владельца станет максимальным из приоритетов ожидающих процессов. И в третьем случае системный вызов просто завершится неудачей. Контроль за оставшимся временем ожидания проводится в уже знакомой нам функции decrease. В случае, когда он становится неположительным, процесс переходит в состояние ENV_RUNNABLE и сразу же запускается в случае, если его приоритет выше приоритета текущего процесса. Функция mutex_unlock, если ее вызывает процесс-владелец мьютекса, понижает приоритет владельца до исходного, пробуждает один из ожидающих процессов(с максимальным приоритетом) и назначает его владельцем. Такая реализация введена потому, что невыгодно пробуждать сразу все процессы, чтобы они конкурировали за мьютекс. Функция mutex_delete уничтожает мьютекс и пробуждает все спящие процессы. Удалить мьютекс может только тот процесс, который его создал. При уничтожении процесса, если он владел мьютексом, он его освобождает. Мьютексы хранятся в массиве. Это создает простор для уязвимостей, и были реализованы некоторые меры предосторожности:
Во-первых, при вызове mutex_create возвращается не позиция мьютекса в массиве(это была бы катастрофа), а некоторое число-код, полученное с помощью функции code.При каждом системном вызове, связанном с данным мьютексом, производится функция decode для определения мьютекса. Далее пользовательские процессы смогут взаимодействовать с данным мьютексом только при помощи этого числа. Во-вторых, один процесс может заблокировать только один мьютекс за раз. В-третьих, сделать unlock может только процесс-владелец мьютекса, а delete-только процесс-создатель. В-четвертых, у каждого процесса есть квота на мьютексы, а иначе злоумышленник мог бы просто в цикле выделить себе все мьютексы и остановить работу системы.

Тесты

Первый тест проверяет корректность очереди мьютекса в случае отсутствия таймаутов. Сначала процессы выстраиваются в очередь заблокированному мьютексу, а затем по очереди завладевают им.
Второй тест проверяет корректность взаимодействия ipc и различных приоритетов. Также проверяется сам механизм назначения приоритета.
Третий тест проверяет корректность мьютексов с ограничением по времени, а также наследование приоритета. Процессы выстраиваются в очередь, далее им назначаются приоритеты и процесс-отец разблокирует мьютекс.
Четвертый тест проверяет корректность некоторых POSIX-функций, а также правильность работы с квантом времени.
Пятый тест проверяет корректность очереди у мьютекса и взаимодействие мьютексов и ipc. Мьютексы выстраиваются в очередь и по очереди начинают взаимодействовать с процессом-отцом.
Шестой тест проверяет корректность наследования приоритета и очереди мьютексов в случае, когда есть несколько процессов с одинаковым приоритетом.
Седьмой тест проверяет корректность работы планировщика в отсутствии активных процессов, а также наследование приоритета в случае с таймаутами.
Восьмой тест проверяет корректность наследования приоритетов в случае с несколькими мьютексами, а также работу планировщика в этом случае. Сначала отец захватывает второй мьютекс, а первый сын-первый. Потом отец назначает первому сыну малый приоритет, а второму сыну-большой и освобождает мьютекс. Таким образом, второй сын должен захватить два мьютекса подряд, а потом первый сын захватит оставшийся мьютекс.