Поддержка многопоточности в пользовательских программах JOS

Алексей Якушев, Денис Кущук, Александр Бойко

1 Реализованные задачи

1.1 SMP

1.1.1 Инициализация ядер процессора

Инициализация происходит при вызове функции mp_init, в ходе инициализации производится парсинг таблицы acpi_rsdp, адрес которой берется из параметров загрузчика. По этой структуре находится адрес таблицы acpi_rsdt, и далее осуществляется парсинг заголовков в этой таблице. Из этих заголовков ищется тот, который относится к APIC. По нему определяется lapicaddr, а также парсятся apic_entry и из них рассматриваются те, что относятся к lapic. По их количеству определяется число ядер процессора. На данный момент JOS поддерживает до 64 ядер. В процессе загрузки работает только один процессор (Bootstrap Processor), поэтому прежде всего нужно запустить остальные ядра AP(Application Processor). Запуск осуществляется при помощи прерываний LAPIC.

1.1.2 Инициализация LAPIC (Local Advanced Programmable Interruption Controller)

Для работы LAPIC используется область размером в одну страницу, замапленную в память по MMIO по адресу lapicaddr. Также в lapic_init определяется локальный для каждого ядра таймер. С помощью функции cpunum можно определить номер текущего ядра процессора.

1.1.3 Запуск ядер Application Processor

В процессе инициализации памяти инициализируются стеки ядер AP, расположенные ниже стека ядра BP. Запуск ядер осуществляется при помощи функции lapic_startap в соответствии с алгоритмом, определенным в MultiProcessor Specification. Ядро запускается в реальном режиме и начинает выполнять код, предварительно помещенный по заданному адресу 0x7000. В этом коде ядро устанавливает стек, временную

таблицу дескрипторов, а также производит переход в 64-битный режим аналогично translation.nasm. Далее управление передается в C-функцию mp_main, в которой производится переключение на адресное пространство ядра, инициализация локального apic и инициализация прерываний. Далее AP сообщает BP о том, что оно запущено.

1.1.4 Блокировка

Синхронизация ядер организована согласно методу big kernel lock. Перед запуском AP BP выставляет глобальную блокировку ядра и ждет, пока остальные ядра запустятся. После запуска ядра AP так же пытаются установить блок и ждут, пока BP его освободит. После того, как все ядра AP запустятся, ядро BP входит в код ядра и выполняет его до тех пор, пока не перейдет на выполенение пользовательского кода. В этот момент BP снимает блок. Далее ядра AP по очереди входят в код ядра и так же снимают блок при переходе в пользовательский код. Каждый раз, входя в код ядра (при прерываниях во время выполнения пользовательского кода), ядро процессора устанавливает блок таким образом, что одновременно код ядра может выполнять только один процессор. Установка блока осуществляется при помощи функции smart_kernel_lock. В этой функции проверяется, был ли установлен блок, и если да, то каким ядром процессора. Так, процессор не будет блокировать сам себя. Также блок снимается в случае, если во время выбора энвайронмента ядро процессора не находит свободных энвайронментов и переходит в Halt. В дальнейшем это ядро может быть активировано с помощью прерывания.

1.1.5 Состояние процессора

Состояние процессора описано в структуре CPU_info. Структура содержит такие поля как id процессора, статус ядра процессора, текущий выполняемый env, текущее адресное пространство и флаг того, находится ли данный процессор в ядре (используется в smart_kernel_lock). Глобальный указатель thisenv заменен на макрос, возвращающий выполняемый env этого процессора.

1.2 Создание нового потока (по аналогии с pthread_create).

Пользователь вызывает библиотечную функцию jthread_create, которая вызывает системный вызов sys_kthread_create. Этот системный вызов делает fork, создает thread environment в рамках вызывающего процесса.

С точки зрения архитектуры физически процессы и треды являются энвайронментами, но логически они разделены между собой с помощью полей в структуре энвайронментов. Поле env_child_thread отвечает за то, является ли энвайронмент трэдом или процессом; в поле env_process_envid находится id процесса, в котором находятся трэды.

Все трэды в процессе используют одно и то же пространство памяти и ту же функцию pgfault. Для каждого из трэдов аллоцируется новый стек; они расположены в памяти подряд один под другим. Здесь же мы отображаем память, отвечающую за Thread Local Storage (TLS). Регистр RSP выставляется на начало стека данного потока. Регистр RIP выставляется в адрес стартовой библиотечной функции, которая запускает функцию, переданную в jthread_create с нужными аргументами, причем эта функция и ее аргументы передаются через регистры RDI, RSI.

1.3 Ожидания завершения работы нового потока (по аналогии с pthread_join)

Функция jthread_join приостанавливает выполнение, пока завершится трэд. Внутри вызывается системный вызов sys_kthread_join, который проверяет, является ли статус трэда "зомби"или "отменен". Функция jthread_join вызывает sys_yield до тех пор, пока трэд не будет зомби или отменен - таким образом, ресурсы не тратятся на простой в ожидании.

1.4 Завершение потока из другого потока (по аналогии с pthread_cancel)

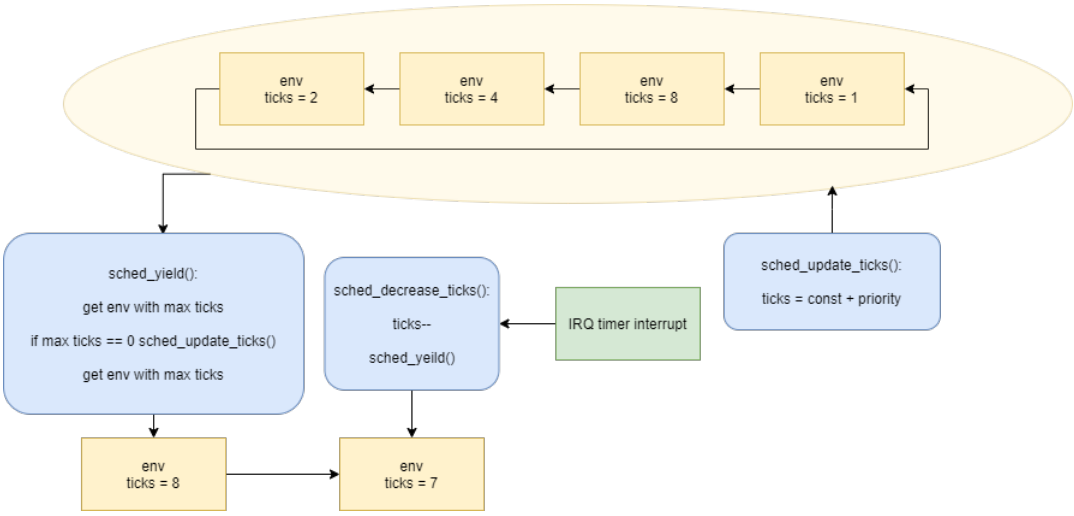
Функция jthread_cancel завершает выполнение трэда, осуществляя системный вызов sys_kthread_cancel. Этот системный вызов выставляет состояние энвайромента в NOT_RUNNABLE, а состояние трэда в "отменен чтобы функция jthread_join поймала отмену трэда. При следующем

вызове sys_yield этот энвайронмент не будет запущен.

1.5 Завершения потока по факту выхода из функции или по вызову функции (по аналогии с pthread_exit)

Как только функция, которая выполняется на треде завершает свою работу, вызывается системная функция sys_kthread_exit, где статус потока переводится в зомби, а статус енваронмента переводится в ENV_NOT_RUNNABLE.

1.6 Планировщик



В качестве планировщика был реализован плоский планировщик с приоритетами выполнения. У каждого энвайронмента есть поле priority и поле ticks. В поле ticks находится число, обозначающее количество "тиков"процессора, которые отведены данному энвайронменту на исполнение. Отводимое количество ticks зависит от приоритета priority. Планировщик при вызове sched_yield выбирает из списка всех энвайронментов тот, у которого наибольшее значение ticks (также учитываются флаги RUNNABLE и другие) и запускает его. При прерываниях от таймера IRQ значение ticks в текущем исполняемом энвайронменте уменьшается на 1, а также вызывается sched_yield. Если все энвайронменты имеют нулевые оставшиеся тики, вызывается функция, назначающая всем энвайронментам новые тики (в соответствии с их приоритетами). Таким образом, достигается справедливое распределение ресурсов.

1.7 Синхронизации между потоками с помощью простейших примитивов синхронизации типа рекурсивных и нерекурсивных мьютексов

В качестве примитива синхронизации был реализован нерекурсивный мьютекс. В структуре мьютекса содержится указатель на владельца мьютекса и булево поле, означающее, что мьютекс заперт или свободен. Функция jthread_mutex_lock запирает мьютекс, если он свободен. Если мьютекс не свободен, вызывается sys_yield до тех пор, пока мьютекс не освободится. Таким образом не возникает простоя ресурсов при ожидании запертого мьютекса. Функция jthread_mutex_trylock аналогична, но возвращается сразу если мьютекст заперт. Функция jthread_mutex_unlock отпирает мьютекс, если владелец мьютекса это тот тред, который пытается отпереть мьютекс.

1.8 Свободного запуска потоков на свободном ядре процессора с поддержкой миграции между ядрами во время работы

Ядра входят в панировщик по очереди, выбирают свободный енваронмент и выполняют его. То есть любой процесс может быть запущен на разных ядрах (за исключением случая, когда мы реализуем binding cpu - там существуют ограничения, привязка процесса к ядру).

1.9 Биндинг потока к указанному ядру процессора (через функцию, доступную пользователю)

Библиотечная функция jthread_setcpu связывает тред с ядром, то есть теперь данный тред может выполняться только на конкретном переданном ядре. Это происходит с помощью системного вызова sys_kthread_setaffinity. Она устанавливает поле affinity_mask в структуре процесса в нужное значение (маску для конкретного cpu). А далее, когда ядро заходит в планировщик для запуска нового процесса, происходит фильтрация процессов, которое ядро может запускать, то есть при помощи поля affinity_mask планировщик определяет можно ли запустить процесс на данном ядре.

1.10 Поддержка Thread Local Storage через ключевое слово _Thread в языке C

В нашей реализации число трэдов может быть не больше чем максимальное число энвайронментов, которое равно NENVS. Так, для аллокации памяти для Thread Local Storage мы отступаем от вершины стэка трэда на NENVS * STACK_SIZE, чтобы эта память точно не затерлась при создании новых трэдов. MASTER_TLS_TOP = USER_STACK_TOP - NENVS * STACK_SIZE.

В функции load_icode, в процессе декодирования двоичного ELF-образа пользовательской программы обрабатывается сегмент заголовка PT_TLS. Обработка происходит таким же образом, как для сегмента PT_LOAD с той разницей, что учитывается выравнивание и сегмент помещается по определенному ранее виртуальному адресу MASTER_TLS_TOP - ph[i].p_memsz. Это мастер-копия thread-safe переменных. Локальные копии мастер-копии располагаются ниже учитывая размер TLS и выравнивание по страницам: ROUNDDOWN(tls_master_mmap, PAGE_SIZE) - ROUNDUP(tls_size, PAGE_SIZE) * threadnum - tls_size

Сразу за локальными копиями TLS следуют самоссылающиеся (требование архитектуры) структуры Struct EnvTLS. В регистр %fs данного трэда помещается адрес этой структуры. Также в структуру TrapFrame был добавлен регистр %fs, сохраняющийся при переключении контекста. При объявлении переменной _thread в коде в ходе работы программы адрес этой переменной определяется как значение регистра %fs минус сдвиг этой переменной. Таким образом гарантируется, что значение переменной не может быть изменено из другого потока.

1.11 Реализация thread-safe errno с добавлением её поддержки к необходимым функциям в соответствии с C и POSIX

thread-safe errno это _thread переменная, своя для каждого из потоков. Она определяется как : _thread int errno. В файле error.h расписаны коды ошибок. Все функции тредов при определенной ошибке записывают ее код в переменную errno и возращают -errno.