CS 452 Kernel Documentation

Prajval Malhotra (p6malhot, 20908293), Sebastian Negulescu (snegules, 20828274)

February 5, 2024

Contents

1 Overview

1.1 Kernel setup

The kernel is located in the repository at https://git.uwaterloo.ca/p6malhot/cs-452.git. The repository title is "CS 452".

To build and run the kernel -

1. `git clone ist-git@git.uwaterloo.ca:p6malhot/cs-452.git`
2. `cd "cs-452"`
3. `git checkout 84c361cd2a8bedd1a75740f3d217238fb476598a`
4. `make` to build the Rock Paper Scissors demo
5. `make PERFORMANCE_MEASUREMENT=on` to build the Performance Measurement demo. It takes the following variables:
   - `OPTIMIZATION=[on, off]` turns compiler optimization (`-O3`) on or off. If this variable is omitted, optimization will be off.
   - `CACHE=[both, icache, dcache, none]` will enable instruction cache (`icache`), data cache (`dcache`), both (`both`), or none (`none`). If this variable is omitted, no caches are enabled.
   - `FIRST=[sender, receiver]` to specify whether the sender or receiver will execute first. Defaults to `sender` when unspecified.
   - `MESSAGE_SIZE=[4, 64, 256]` to specify the message size. Will default to 4 bytes when unspecified.
6. Upload `kernel.img` to Raspberry Pi.
7. When rebuilding the image with different parameters, run `make clean` before building to get rid of the old image or the new one will not get created.

1.2 Data Structures Used

1.2.1 Task Descriptor

Relevant files - `src/task.cc` `include/task.h`
Our task descriptor consists of the following fields -

- `struct SenderQueue`
  - `TaskDescriptor *head` - a pointer to the head of the queue
  - `TaskDescriptor *tail` - a pointer to the tail of the queue
- `struct SenderParameters`
  - `int name_request` - if we are providing a request to the name server, we fill out this field with `RegsiterAs` or `WhoIs`
  - `const char *msg`
  - `int msg_len`

  - `char *reply`
  - `int reply_len`

- `struct ReceiverParameters`
  - `int *tid`
  - `char *msg`
  - `int msg_len`

- `uint8_t stack[STACK_SIZE]` - a stack, of size STACK_SIZE
- `volatile uint8_t *stack_ptr` - the tasks stack pointer
- `int tid` - tid of the current task
- `int parent_tid` - tid of the current task's parent
- `int priority` - the priority of the task
- `TaskState state` - the state of the task (either running, send blocked, or receive blocked)
- `void (*function)()` - the task function
- `TaskDescriptor *next` - a pointer to the next free task descriptor in the free list.
- `SenderParameters sender_params` - space to store sender parameters in case of block
- `ReceiverParameters receiver_params` - space to store receiver parameters in case of block
- `SenderQueue sender_queue` - a queue to hold blocked sender tasks
- `TaskDescriptor *next_sender` - a pointer to the next sender task in the queue
- `function int add_sender_task(TaskDescriptor *task)` - pushes a sender task to the blocked queue
- `function TaskDescriptor *pop_sender_task` - pops a sender task from the blocked queue and returns a pointer to it
- `function init_task` - a function to initialize the task descriptor and registers.

1.2.2 Kernel data

The idea behind this struct was to remove global variables.

- `int global_tid` - holds the next tid to be allocated, gets incremented each time a new task is created
- `TaskAllocator *task_allocator` - a pointer to an object of the `TaskAllocator` class, used to allocate space for new tasks.
- `Scheduler *scheduler` - a pointer to an object of the `Scheduler` class, used to schedule new tasks.
- `TaskDescriptor *task_list[]` - an array of size `MAX_SIZE` storing pointers to `TaskDescriptor`s. This array keeps track of `TaskDescriptor`s by their `tid`. `MAX_SIZE` has the current value of 128, supporting 128 tasks.

2 Scheduler

Relevant files - `src/scheduler.cc` `include/scheduler.h`

- The scheduler follows a highest priority first and then round robin selection method for selecting the next task to execute.
- There are a total of 10 priority levels, 0 being the highest (will execute first), and 9 the lowest (will execute last).

2.1 Data Structures Used

Our choice of data structures was a `Scheduler` class with a nested `ReadyQueue` struct.

2.1.1 class Scheduler

The contents of the class are -

- A nested structure, `ReadyQueue`
- An array of `ReadyQueue`s who's size is defined by the macro - `NUM_PRIORITIES`
- A pointer to a `TaskDescriptor`, `current_task` to refer to the current executing task - this is used throughout the kernel to refer to the current/last executing task when control returns back to the kernel from an exception.
- Functions to implement the scheduler

2.1.2 struct ReadyQueue

- a struct that contains pointers to two `TaskDescriptor`s, which point to the head and tail of the TaskDescriptors currently on the ready queue
- Here, head is the next, and tail is the last task to be scheduled at that priority level.

2.2 Implementation

2.2.1 `add_task`

Adds a task to the relevant priority level `ReadyQueue`.

Return values -

- `-1` if the priority level of the task is invalid.
- `<priority level>` of task if it was added successfully.

Argument(s) -
`TaskDescriptor *` - the task to add to the scheduler.

2.2.2 `schedule_task`

This function does the actual scheduling. It is called from `mainloop` upon returning from a user task, to first remove the current running task, then add it back to the Ready Queue and finally, find the next task to schedule based on a priority, round-robin scheduling method.

Return values -

- `-1`: No more tasks to schedule.
- `0`: `current_task` set to a task to be executed next successfully.

3 Task Allocator

Relevant files - `include/task_allocator.cc`, `include/task_allocator.h`

- We use a combination of slab allocation, freelists, and intrusive lists to implement the task allocator.
- The task allocater is allocated about 520kb of memory to assign to tasks, starting from the end of the bss section and is 16-byte aligned.
- This amount of memory is defined in the linker and allows for around 128 tasks with 4kb stack sizes to exist.

3.1 Data Structures Used

We use a `TaskAllocator` class, which contains a `Slab` union

3.1.1 class `TaskAllocator`

- A union, `Slab`
  - `Slab * next` - a pointer to the next free Slab
  - `TaskDescriptor td` - The `TaskDescriptor` of the current task using the slab
- `Slab * free_list` - a pointer to the first free slab
- Functions to implement slab allocation

3.2 Implementation

3.2.1 `allocate_task`

Called to allocate a slab for a task. Gets a pointer to the head of the freelist, which points to the first free slab, casts to a `TaskDescriptor` and returns the pointer. Also updates free list pointer to point to next free slab.

Return Values -

- Returns a `TaskDescriptor *` if free list has available slabs to allocate.
- Returns a `nullptr` if there are no free slabs in the freelist.

3.2.2 `free_task`

Called to free a slab. Gets a pointer to the task to free, casts it to a `Slab *` and adds it to the start of the freelist.

Argument -

- `TaskDescriptor *task` - the task to free.

4 System Calls & Exception Handling

4.1 System Calls

Relevant files - `src/kernel.cc`

- The user triggers a system call, which in turn makes an SVC call with the appropriate code, which is then handled by the exception handler.
- The exception handler as described in the exception handling section returns back to `mainloop` which runs `process_request` to process the request made by the user task.

4.1.1 process_request

Processes any SVC calls issued by user tasks.

Task Creation System Calls

- `Create` - Calls `allocate_task`, sets up the new task descriptor and adds it to the scheduler. Puts the created task's tid in register `x0` as a return value if the task was created successfully. If not, then -1 is put into register `x0` to signify an invalid priority argument or -2 is placed in register `x0` to signify that there are no more task descriptors that can be used.
- `MyTid` - Put the current tasks tid in register `x0` to interpret as the return value.
- `MyParentTid` - Put the current tasks parent tid in register `x0` to interpret as the return value.
- `Yield` - Simply return back to the kernel since we anyways will reschedule and decide on a task to run next.
- `Exit` - Calls `free_task` and sets `current_task` to `null`

Message Passing System Calls

- `Send` - Searches for the receiving task with the provided `tid`. If the target task cannot be found, -1 is returned to the calling task. Otherwise, if the receiving task is currently blocked and waiting to receive, then the sending task's message is copied into the receiver buffer and the number of bytes copied is returned to the sender. If the receiver is the name server, then the caller must be making a name server system call, and the request type is copied to the receiver buffer. The receiving task is then unblocked. However, if the receiving task is not currently blocked, then the sending task gets added to its sender list and becomes blocked.

- `Receive` - Receives a message. This function first checks if there are any sender tasks blocked on it's sender list. If there are no tasks blocked, then it will store it's parameters into a structure and block until a sender targets it with a message. If there is at least one task blocked on it's sender list, then it will remove it from it's sender list, and copy the sender's message buffer into its own and the sender's `tid` into the provided `tid` reference. If the receiver is the name server, then the sender's `name_request` is copied into the beginning of the receiver's message buffer.
- `Reply` This function sends a reply message from the receiver to the blocked sender. The function copies the reply from the receiver to the sender and unblocks the sender. If the function cannot find the sender based on the given `tid`, then -1 is returned to the receiver. Otherwise, the number of characters copied is returned to the receiver.

Name Server System Calls

- `WhoIs` - A wrapper for send, where the tid becomes that of the name server, the message becomes the name plus a byte for the request type (`WhoIs` in this case).
- `RegisterAs` - Also a wrapper for send, the tid is that of the name server, and the messages is the passed in name plus a byte for the request type.

Clock Server System Calls

- `Time`
- `Delay`
- `DelayUntil`

Synchronization System Calls

- `AwaitEvent` - Takes in the event ID the task is requesting to wait on as an argument. Based on the event ID, it checks if the event has already occurred, in which case it puts the task back into the ready queue. If not, the calling task's state is changed to `EVENT_BLOCKED` which will eventually be changed to `READY` when the appropriate event occurs.

  The function returns an int representing status, 0 for success, and -1 in case of any errors. This can be modified in the future as per requirements.

  A new class was added to store data related to waiting tasks and events.

4.2 Exception Handling

Relevant file - `src/exception_vector_table.S`

- We assign `VBAR_EL1` in boot.S to the exception vector table's address, which is located in the text section.
- Upon receiving an exception from an `SVC` call, `syscall_to_kernel` is called, which does a chunk of the exception handling and eventually returns control to the kernel loop.

4.3 `syscall_to_kernel`

- Called by low level exception handler upon receiving an SVC call
- Switches to user stack, and saves caller-saved registers, and the SP on the stack, along with information about the exception, and then switches back to the kernel stack.
- Back on the kernel stack, the user stack pointer is saved, and `SPSR` is restored.
- The rest of the kernel registers are then popped form the stack.
- Register `x0` stores the SVC call code, used to determine the type of exception to handle. `x0` is then interpreted as the return value of the exception handling function once back in the kernel.

5 Interrupt Handling

Interrupt handling is done in a similar way to exception handling and system calls, where the process begins in the `exception vector table`.

- When an IRQ interrupt occurs, the program jumps to the `exception vector table` and then branches to the `interrupt_to_kernel` handler.
- In the handler, the current task's state is saved, and the kernel's is loaded. A special number `0x100` is placed into the return register for the kernel. The handler returns to the kernel to process the request.
- The kernel, now back in the main loop, calls `process_request` with the value `0x100` to handle the interrupt.
- In `process_request`, the register `GICC_IAR` is read to get the interrupt ID.
- If the interrupt ID is that of timer `C1` (97), then the timer is set to 0 and the interrupt is cleared.
  - If a task is not waiting on the interrupt, a flag is set to cache the interrupt in case a task comes along.
  - Otherwise, the waiting task gets unblocked and added back to the scheduler.
  - The timer is reset to its next 10ms target.
- If the interrupt ID is that of timer `C3` (99), then the timer is set to 0 and the interrupt is cleared.
- The kernel then exists `process_request` and schedules the next task as usual.

6 Context Switching

Relevant files - `src/bootstrap.S`, `src/task.cc`

Upon returning from a user task, control always returns back to the scheduling loop in `mainloop`. First, the user task's svc call is processed, and then we loop back around, schedule a new task and initiate the context switch, which is triggered by the function `switch_to_task`.

6.1 `switch_to_task`

This is an assembly function, responsible for context switching from the kernel to the user task.

- We first set `SP_EL0` to the user stack pointer passed as an argument so that we don't lose it.
- Next, we save the kernel caller-saved registers along with necessary system registers.
- After this, we switch to the user stack and pop the user registers off the stack.
  - Notably, we pop the exception link register into `x0`, which is the address (of the task function) we return to when we `eret`.
- Finally, we return from the exception using an `eret`. Note: this changes the program exception level from EL1 to EL0.

Arguments -

- `x0` - user task stack pointer
- `x1` - pointer to user task stack pointer

6.2 `bootstrap_task`

This is the function responsible for actually executing task functions. It sets up the user stack pointer, reserves space, saves the EL0 stack pointer, sets the exception link register, the saved program status register, switches back to the kernel stack and finally, returns.

Arguments -

- `x0` - user task stack pointer
- `x1` - task function pointer
- `x2` - pointer to user task stack pointer

6.3 Task execution

The first user task is initialized in `mainloop`, and all subsequent tasks are initialized in `process_request`. This is done by making a call to a task initialization function, which 1. initializes the task descriptor, and 2. calls `bootstrap_task` to set up the task's state such that we can simply context switch to the task in the future if we want to execute it - even if it has never been executed before.

7 Name Server

Relevant files - `src/name_server.cc`

The name server is a task that keeps track of other task's names. It does this using a `name_map`, which is just an array of `name_pairs`. A `name_pair` contains a character array with the name of the task, and an integer representing the tid of the task. The name server runs a continuous for loop which executes the following tasks:

1. `Receive` a request (either to register a name or find a name).
2. The first byte of the request will be its type, the rest will be the name to register or find.
3. If the request type is to find a name, the name server will loop through it's map of names and compare them to the name to find. If that name exists in the map, then its corresponding tid will be gotten and placed into the first byte of the response. There are such a small number of tasks that we are not concerned with truncating the returned tid. If that name does not exist in the map, -1 is placed in the first byte of the response, and the response is `Reply`'d to the inquiring task.
4. If the request type is to register a name, the name server will first check if that tid already has a name. If it does, then the name is overwritten with new name in the same entry in the `name_map` and 0 is sent in a `Reply`. Otherwise, the name server then checks if that name already has a tid associated with it. If it does, then the tid is overwritten with the new tid in the same entry in the `name_map` and 0 is sent in a `Reply`. If neither the name nor the tid are already in the name server's map, then a blank entry is found and the pair is entered there and 0 is sent in a `Reply`. If a blank entry cannot be found then -1 is sent back to the inquiring task signifying there are no free slots.

This task never terminates, and will live for the lifetime of the kernel.

8 Clock Server and Clock Notifier

Relevant files - `src/clock_server.cc`

The clock server and notifier are responsible for maintaining system time. The clock server is a high priority task that keeps count of the number of 10ms ticks that have occurred since the task has been initialized. It is the parent of the clock notifier task, an even higher priority task who's responsibility is to wait for a system timer interrupt, and then notify the clock server a tick has happened. The clock notifier repeats this process indefinitely. The clock server also keeps track of tasks waiting for a delay, blocking them until that delay has been reached. It does this with an integer array `delay` of size `MAX_TASKS` to hold the delay in ticks of every task, and a boolean array `delay_map` of size `MAX_TASKS` to indicate whether that task has an active delay. The clock server runs on a continuous loop, executing the following tasks:

1. `Receive` a request (either a clock tick, a time request, a delay request, or a delay until request).
2. Extract the first byte of the request to determine its type.
3. If the request is a clock tick, then the server increments its tick counter, `Replies` to the clock notifier, and loops through the `delays_map` array. If any flags are set, then the server checks if the corresponding delay in `delays` is less than or equal to the tick counter. If the condition is true, then the server sends a `Reply` to that delayed task and resets its flag in `delays_map`.

These tasks never terminate and will live for the lifetime of the kernel.

9 Known Bugs and Unimplemented Features

- The `Send` system call will not return -2, as there is no case that is handled that would result in the send-receive-reply transaction not being able to be completed. Currently, the behaviour is assuming that all `Send`s will be `Reply`'d to and unblocked eventually.
- This brings up a bug where if two tasks `Send` a message to each other, then they will be indefinitely blocked and deadlocked, each waiting for a response that will never come...
- Another bug is a part of the rock paper scissors server. It only supports up to 20 clients at a time, which is more than the number of tasks we can have at this time.
- Furthermore, not much mediation is done to protect the client from inputting an incorrect sequence of instructions to the server, such as registering twice before quitting once. By registering multiple times before quitting, the server will not be able to distinguish the proper game to play.

10 Measurements

10.1 Performance Measurement

Relevant files - `src/srr_performance.cc`

We measured our program's send-receive-reply circuit performance with 15 rounds of 10000 iterations each. Each round would start by getting the current time, and then looping 10000 times either just executing `Send` as the sender or `Receive` and `Reply` as the receiver. At the end of the 10000 loops for the round, the time is taken again and then subtracted by the time at the start of the round. That difference is then divided by 10000 to get the average time taken per send-receive-reply loop, and then logged to output. At the end of the 15 rounds, each averaged time is then averaged again by the number of rounds to get the total average time per send-receive-reply loop.

At the start of the test, we measure the time it takes to read the clock, that usually takes around 5 microseconds. Since we only measure the clock twice, it is a negligible amount over 10000 loops.

One flaw in the measurement process has to do with the looping nature. After the `Reply`, a context switch will occur outside of the cycle, but will be counted in the measurement 10000 times. This will lead to slightly inflated results as we are secretly timing an extra context switch every loop.

The results gotten were about as expected.

For the message size, all measurements scale approximately linearly with the number of bytes in the message, no matter the other parameters. There was some slight variation, but by taking the time between 4 and 64 byte measurements, we can get the time to fill 60 bytes. Scaling this to 252 bytes and adding it to the measurement with 4 byte messages, we get approximately the 256 byte measurement time. This shows a linear relationship between the amount of bytes in a message and the time taken.

In general, the tests with the sender going first took a few microseconds longer than the tests having the receiver going first. This can be explained by the fact that the receiver does not need to be pushed or popped to/from any queue when it blocks, while the sender

does. Avoiding this operation can save a bit of time when the receiver blocks rather than the sender.

The instruction cache offered a much greater benefit than the data cache. Enabling just the data cache offered a negligible improvement over no caching of just a microsecond or two, explainable by variance. The instruction cache gave a boost of around 5 microseconds in most cases, the same as both caches being enabled.

The optimization being enabled gave the biggest boost to performance. It more than halved the amount of time in general for small message sizes. As the message size grows, so do the performance savings from optimization. The biggest benefit is given to the 256 byte message sizes, with times taking a quarter of what they used to.

10.2 Idle task Measurement

The idle task is created by the first user task with the lowest possible priority(currently 9). We measure the idle task time against the overall task time.

Overall task time is calculated in the kernel by storing the start time before running a task and end time right after we receive an interrupt, or exception. This method ensures that we exclude as much of the kernel processing time as possible and have a more accurate representation of idle task time.

Idle task time is computed similarly, except we only store the start time if the next task to be run is the idle task.

This information is then processed in the interrupt and syscall handler, is printed only if there was a change of at least 1%.

The idle task itself is an infinite for loop, which calls wfi in every iteration.

We created a new class to store metadata related to task timings.

Our observation was that idle task initially starts off fluctuating a bit, with a relatively lower value (∼70%) eventually rises up to 94% as the tasks finish executing, and over time goes up to 99% if the program is left to run. This is expected, given that initially, the other 4 tasks run as well. When each task finishes, the idle task time as it accumulates, goes up.