Kernel 1 Specs - Benjamin Zhao Part

Tid

Tids are the unique identifiers for tasks tracked as unsigned 32-bit integers. Every task has a unique tid, and currently all zombie tasks return the rid upon exiting. The upper 16 bits of the tid represent a version number and the lower 16 bits represent an id. All tids start at version 0.

For example, tid `0x0000|0001` represents a task with id `1` and version `0`.

`Tid: 0x 0000 | 0001`

We can mask the lower bits to obtain the id: `Tid_id = Tid & 0xffff` We can shift the upper bits and mask to obtain the version: `Tid_ver = (Tid >> 16) & 0xffff`

TidTracker

The TidTracker is a distributer which distributes unique tids upon request. The tids are pre-generated when the kernel starts running. Tids are re-used (with an incremented version) when a task exits and the Tid is returned to the distributer for re-use. When the distributer runs out of tids, likely either two things have happened: - All tids are in use - The use of a tid has exceeded

$$2^16$$

re-issues in which an overflow may cause undefined behaviours

The TidTracker uses a circular buffer, prefilled with some maximum number of Tids allowed to be allocated to tasks at once. When a task is created, the kernel requests for a Tid using tt_get() from the TidTracker. The tracker then takes the first tid from the queue, pops it and gives it to the kernel. When the task exits, the kernel calls tt_return to return the tid to the tracker. The tracker appends 1 << 16 to the tid, and inserts to the end of the buffer.

```
int tid = tt_get(&tid_tracker);
tt_return(td->tid, &tid_tracker);
```

The circular buffer is implemented using a fixed-sized array, with a start and end index pointing to the head and tail of the queue respectively. The queue takes constant O(1) insertion and deletion of head. The circular buffer itself does not have any overflow guards, however we rely on the limited number of tids to ensure we never reach an overflow.

```
typedef struct CircularBuffer {

  unsigned int buffer[CIRCULAR_BUFFER_SIZE];
```

```
  unsigned int buffer_start;
  unsigned int buffer_end;

} CircularBuffer;

void init_circularBuffer(CircularBuffer *buffer);
void push_circularBuffer(CircularBuffer *buffer, unsigned int val);
unsigned int top_circularBuffer(CircularBuffer *buffer);
void pop_circularBuffer(CircularBuffer *buffer);
```

Context Switch

A context switch is the action of changing between user tasks and kernel. A task can give up control and return to the kernel through system calls. It is the kernel's job to ensure that the task cleanly changes back to kernel state by saving its states onto the user task stack. A kernel will give up control once it has handled any syscalls that have been made by a previous task and resumes the next highest priority task through scheduling. The kernel must preserve the state of itself and restore any states the active task may have.

The `activate` function does the following (Kernel -> User):

```
 //Store Kernel State
  PUSH_STACK("r0-r12, lr");
  //Push ret val to stack as temp
  asm("mov r8, %0"::"r"(td->ret));
  PUSH_STACK("r8");
  //Install SPSR
  WRITE_SPSR(td->psr);
  //Change to system mode
  SET_CPSR(SYSTEM_MODE);
  //Change the stack pointer to the task's stack (uses fp so no worries)
  WRITE_SP(td->sp);
  //Load instruction after swi (r4) from user stack
  POP_STACK("r4");
  //Change to kernel mode
  SET_CPSR(KERNEL_MODE);
  //Save into kernel lr for loading
  asm("mov lr, r4;");
  //Change to system mode
  SET_CPSR(SYSTEM_MODE);
  //Load the User Trap Frame
  POP_STACK("r0-r12, lr");
  //Switch back to kernel mode
  SET_CPSR(KERNEL_MODE);
  //Set r0 with the new return value from stack
```

```
  POP_STACK("r0");
  //Move to the user task
  REVERSE_SWI();
```

Upon return to the activate function via SWI (User -> Kernel):

```
//AFTER USER TASK CALLS SWI (CANT USE FP)
  asm("KERNEL_ENTRY:");
  //Change to System mode
  SET_CPSR(SYSTEM_MODE);
  //Save the user state
  PUSH_STACK("r0-r12, lr");
  //Change to Kernel mode
  SET_CPSR(KERNEL_MODE);
  //Save lr to stratch r3
  asm("mov r3, lr");
  //Change to System mode
  SET_CPSR(SYSTEM_MODE);
  //Save the lr(r3)
  PUSH_STACK("r3")
  //Change back to kernel mode
  SET_CPSR(KERNEL_MODE);
  //load the kernel stack (fp is now resuable again!)
  POP_STACK("r0-r12");
  //Change back to system mode
  SET_CPSR(SYSTEM_MODE); //Note we can still use fp!
  //Save the user sp to TaskDescriptor's sp
  READ_SP(td->sp);
  //Change back to kernel mode
  SET_CPSR(KERNEL_MODE);
  //Save the spsr to the TaskDescriptor's psr
  READ_SPSR(td->psr);
  // manually put swi arg in r0, avoid overhead of return
  SWI_ARG_FETCH("r0");
  //Restore link register to return properly
  POP_STACK("lr")
```

The activate function runs a set of inline assembly macros which perform the saving of states to stacks, register manipulation, prviledge changes and jumps to and from user land. A limitation of the context switch, setting the CPSR to change mode requires the use of r12, which the context switch does not save.

Syscalls

The kernel supports the following syscalls:

- Assert - Invoked via `assert` provides a method of testing conditions in tasks
- Create - Creates another task to be put on the kernel's task schedule
- Get Tid - Get the task's tid
- Get Parent Tid - Get the parent's task tid
- Pass - Give control away
- Exit - Become a zombie

A user task may call a syscall at any moment during the lifetime of the user task, however it may not be guaranteed that the syscaller will retrieve control right after a kernel finishes handling the syscall. In which case, the return value of the syscall is stored into the TaskDescriptor and is later loaded into the user state through activate. A function in the kernel called handle takes the argument from the task's syscall to determine which request the user task made.