Kernel context

The user context described in the previous section is not the state of the CPU as the kernel runs, but the state of the CPU at the time the trap occured, while the user-level process was executing. In invoking the kernel, that state has disappeared, at least partially. The part of real operating systems that saves this context without trampling over it is usually a very carefully crafted sequence of instructions written on assembly. This is because extraordinary control must be exercised to ensure that, in saving a register to memory, another register that may not have yet been saved is not overwritten. Lucky you, you need not worry about this. The context in the `UserContext` structure passed to your trap handlers contains the consistent state of the cpu at the time immediately before the trap occured. Also, the contents of the same `UserContext` structure will be consistently restored to the CPU when your trap handler returns.

Your kernel also has context, which must also be saved and restored. Think of what happens when a process wants to write to a terminal. The process starts by issuing a `TtyWrite` syscall, which causes a `TRAP_KERNEL` trap, which invokes the `TRAP_KERNEL` handler. After some tortorous code, the kernel talks to the terminal device via `TtyTransmit`. And what does the kernel do next? The output will take a long time to complete. Having the kernel busy-wait would not only be extraordinarily inefficient but would also be fruitless, since the kernel is not interruptible and, therefore, would never acknowledge the interrupt that would signal the output's completion. We know therefore that the kernel must return to user mode, after appropiately blocking the calling process by placing its PCB in some terminal queue. The fact that the kernel would be waiting, and the context of the wait, must be remembered. This context, while related to the user context that caused the `TRAP_KERNEL` (`TtyWrite`) trap, is not the same: it has to do with devices, queues and PCBs rather than with strings. We need to save this context (the memory of what the kernel was doing on behalf of our process,) somehow, and restore it later when the terminal interrupt finally arrives.

This kernel context is saved into a structure of type `KernelContext`. Just like the state of the user process is not only the `UserContext` structure, but is also the virtual address space which must be activated and deactivated in concert with the `UserContext`, the state of the kernel's execution is not only the CPU state contained within a `KernelContext` structure but also resides in the kernel's stack. In our scenario above, then, we would save the kernel stack's virtual memory map and the kernel's CPU registers in the PCB of the process, and reactivate them once the interrupt arrived.

But not so fast. This is actually a little involved. How can the kernel, running on the CPU, save the state of the CPU consistently, atomically? Will we have to write a carefully crafted sequence of instructions in assembly? Well, yes, but we and Solaris have done it for you, to a large extent.

The difficulty is that to manipulate the stack and registers on which the kernel is running we cannot be running on those registers and on that stack while we do it, or we have to do it carefully. We provide you with the function `KernelSwitchContext` which calls a procedure you specify. Within that procedure, but only within that procedure, you can be as sloppy as you'd like and deactivate one kernel stack and activate another without worrying about bootstrapping, though you still have to be careful while realizing that any data on one kernel stack won't be available after you blow that stack away. Here is the prototype of `KernelSwitchContext`:

- `int KernelContextSwitch(KCSFunc_t func, void *, void *)`

The type `KCSFunc_t` (Kernel Context Switch Function) is the type of your procedure wherein you can manipulate the kernel context.

The prototype of a procedure of type `KCSFunc_t` is:

- `KernelContext * func(KernelContext *, void *, void *)`

  The two `void *` arguments passed to `KernelContextSwitch` will be passed unchanged to `func`. The first argument of `func` is the context of the CPU at the time `KernelContextSwitch` was called. The `KernelContext` returned by `func` will be activated after `func` returns.

For example, if your `func` was something like:

```
KernelContext * func(KernelContext * old, void * p1, void *
p2) {
  PCB * OldPCB = (PCB *) p1;
  PCB * NewPCB = (PCB *) p2;

  memcpy(OldPCB->kctxt, old, sizeof(KernelContext));
  <write PT0 entries with entries of K stack in NewPCB>
  <Flush K stack addresses from TLB>
  return New->kctxt;
}
```

then the follwing pseudo-code sequence would block the current processes until some other part of the kernel called `KernelContextSwicth` at a later time, when `BlockPCB` would be in the role of `ReadyPCB`:

```
...
<Here, BlockPCB is active and running>
mempcy(BlockPCB->uctxt, uctxt, sizeof(UserContext));
Enqueue(BlockPCB, DeviceQ);
ReadyPCB = DeQueue(ReadyQ);
KernelContextSwitch(func, (void *) BlockPCB, (void *)
ReadyPCB);
<deal with device completion>
```

```
memcpy(uctxt, BlockPCB->uctxt, sizeof(UserContext));
....
```

After `BlockPCB` was later moved, by some interrupt, to the `ReadyQ`, and it was chosen as `ReadyPCB` was chosen above, the execution would continue at the line `<deal....`

You can also use `KernelContextSwitch` to extract the kernel context and make a copy of the current kernel stack for a new process, without switching to another one. You will probably need to do this within your implementations of `KernelStart` and `Fork`.

This may all seem rather complicated to you right now, but it will get easier. Keep trying to wrap your head around it.

Note that the saving and restoring of the user context above may be unnecessary, depending on how you implement your kernel. Note that, if instead of storing the process' user context in the PCB you leave it on the kernel stack where your trap handler received it, it will be automatically saved and restored when you save and restore the process' kernel context.

Finally, you may be wondering why the `UserContext` and `KernelContext` structures look so different, even though they are both stores for CPU registers. The answer is that the `UserContext` structure is simplified. We have made that one simpler because it is the one where you will be reading and writing registers such as the program counter and the stack pointer. The registers within the `KernelContext` can remain opaque to you and `KernelContext` may therefore retain its full glorious complexity.