

Tutorial 7

Modifying Simple Kernel

1 Some Simple Changes to Simple Kernel

This exercise is unassessed, however, the results of this exercise will be needed for the assessed exercise which will be issued shortly. The objective is to get you started at editing the Simple Kernel code.

1.1 Important

Take a fresh copy of the Simple Kernel source code by typing

```
cp ~ob3/SimpleKernelExercise2.tar.gz .
```

and then unpack this as before.

1.2 Keeping Track of Process State

Simple Kernel offers no easy way of finding out what a processes' current state is — the only way would be to search the ready queues and delay queue. If it is not running, not ready and not delayed, then it has to be suspended. The idea here is that we are going to maintain an extra field in the Process Control Block (PCB) that indicates the state of a process. All we have to do is make sure we change that on all state transitions.

- Add a type declaration for process state to `procP.h`, as follows:

```
15 typedef enum {
16     RUNNING ,
17     READY ,
18     SUSPENDED ,
19     DELAYED
20 } state;
```

- Add a field `s` of type `state` to the process control block (also in `procP.h`).
- Add a function prototype `change_state` in `procP.h`, as follows:

```
65 void inherit_priority( const int producer , const int consumer );
```

- Implement the function `change_state` in `proc.c`.
- Find the source code for all kernel functions which can cause a process state transition, and insert a call to `change_state` at the end of the mutual exclusion region, changing the recorded state of the process appropriately.
- *For Verification*
Get processes to print their state transitions for some selected methods, such as `setready`. You'll find `con_outs` (console output string), which takes a string parameter, useful for this.

1.3 PIDs

Most operating systems (e.g. Linux) maintain a PID (process identifier), a unique integer that identifies a process. You can see these for example by running either the `ps` or the `top` command under Linux. Simple Kernel does not have PIDs. Your task here is add this feature.

- Change the definition for the process control block (PCB) in `procP.h` to include an additional field `int pid`.
- Add a global variable `int next_pid = 0;` to `proc.c`. As indicated, this is initialised to zero.
- Modify the prototype for `create` in `proc.h` to return an integer — this will be the PID.
- Modify `create` in `proc.c` to allocate each process a unique process identifier, which is stored in the PCB's `pid` field.
- *For Verification*
Get processes to print their priority in the `setready` method. You can use `con_outdec`, which takes an integer parameter, for this.

1.4 Maintain a Process Lookup Table

Having PIDs to identify processes is very useful; however, if one wants to use a PID to make a changes in a processes' PCB, one might still end up having to search for the correct PCB pointer, for example if the process is `READY`. Your task here is to maintain a simple lookup table that allows us to find the PCB pointer for a specific process.

- We will assume for simplicity that there is a maximum number of processes. Create a table of PCB pointers in `proc.c`, as follows:

```
22 #define MAX_PIDS 1024
23 process processesTable[MAX_PIDS];
```

This table will be indexed by PID, i.e. `processTable[i]` is the pointer to the PCB of the process with PID `i`.

- Add code to `create` which records the pointer to a newly created PCB in the `processTable`.

Notice that because processes do not exit, and all processes are created at system start, the pointer corresponding to a particular PID cannot change, therefore the above changes are sufficient.