Assignment 4

Question 1

Storing the processor state in the process table instead of on the process stack is an approach akin to Linux's (and other OS's) process control block structure - instead of saving process information inside that process' stack, the information is saved within the structure representing the process in the process table.

With this implementation, the procent struct would look something more like this:

```
struct procent {        /* Entry in the process table       */
  uint16    prstate;    /* Process state: PR_CURR, etc.     */
  pri16    prprio;     /* Process priority         */
  char    *prstkptr;  /* Saved stack pointer          */
  char    *prstkbase; /* Base of run time stack       */
  uint32    prstklen;   /* Stack length in bytes        */
  char    prname[PNMLEN]; /* Process name             */
  sid32    prsem;      /* Semaphore on which process waits */
  pid32    prparent;   /* ID of the creating process       */
  umsg32    prmsg;      /* Message sent to this process     */
  bool8    prhasmsg;   /* Nonzero iff msg is valid     */
  int16    prdesc[NDESC];  /* Device descriptors for process   */
  uint32 stacktab[14];
};
```

where stacktab contains enough elements to store each process and hardware register, the coprocess status, and the stack pointer stack pointer.

When `ctsw()` is called in this implementation, instead of pushing and popping values to and from process stacks, it will simply put them into and get them out of the indices of `stacktab[]`:

```
for (i registers):
  stacktab[i] = register[i];
end for;
```

```
for(i registers):
  register[i] = stacktab[i];
end for;
```

In practice, those for loops would likely be replaced by a series of statements for each index of `stacktab[]`:

```
stacktab[0] = r0;
stacktab[1] = r1;
stacktab[2] = r2;
stacktab[3] = r3;
...
stacktab[12] = lr;
stacktab[13] = cpsr;
stacktab[14] = sp;
```

and

```
r0 = stacktab[0];
r1 = stacktab[1];
r2 = stacktab[2];
r3 = stacktab[3];
...
lr = stacktab[12];
cpsr = stacktab[13];
sp = stacktab[14];
```

The advantage of this implementation is clear from the point of view of a debugging developer: the array is a much cleaner conceptual implementation in many ways as compared to the process stack. Putting elements back into their respective registers when resuming a process, while not necessarily *actually* simpler than when using a stack, may be conceptually easier to understand. From the point of view of the operating system, moving this information off of the stack and into memory puts the information in a less potentially volatile location/more protected location than the process stack.

On the other hand, this requires enough extra memory allocated per procent to hold all the register values and process information, instead of that information being stored on the already-allocated process stack. In this aspect, storing the process information on the process' stack has a distinct advantage, as you need at least 56 (13*4) extra bytes of memory per procent.

Question 2

To change the implementation of killing the current process, I made changes in 2 files: `kill.c` (of course), and `create.c`.

```
```

newpid() `in` create.c `then finishes the job of freeing the stack and assigning` PR_FREE `, because` newpid() `searches through` proctab[] `for those` PR_FREE `(or, now,` PR_DYING `) statuses. Because it searches through` proctab[] `anyway, adding an extra step to identify` PR_DYING `processes, freeing the stack, and assigning` PR_FREE``` to that process is fairly straightforward.

Question 3

`resume()` needs to collect the process priority before calling `ready()` because there is a chance that another, higher-priority process will run before the process being handled currently, that other processes will run in arbitrary fashions, possibly killing or otherwise modifying the process that was originally called. The priority could change (or, in the case of killing the process, disappear), throwing the scheduling invariant off.

The command `badprio` is written (in theory) to demonstrate how `resume()` may return a priority value the resumed process never had, even after resumption. I haven't been able to get this working incorrectly...correctly thus far, but the idea behind the position of `prio`'s assignment is that, as I said above, `resume` needs to know that the priority it is reporting is the true priority of the process *at the moment the process is ready to run*. This is important because, as said above, once `ready()` is called, any process in the ready list may run, assuming their priority is higher, and that arbitrary process may change any number of things on the ready list and process table, including the existence of the original process upon which `resume()` was operating.

To do this I simply moved the assignment of `prio` in `resume()` 2 lines down to the line after the call to `ready(pid)`. `badprio` then creates child processes of higher priorities than itself (and one with a higher priority than the other) in an attempt to get the lower priority process' `resume`. I say attempt because it fails to work incorrectly, and I'm not entirely sure why - any combination or order of priority or process calling/resumption does not seem to have any discernible affect on the priority of any process. Changing up what the child processes do does not affect anything either. I feel like I'm missing something fairly basic in what I'm doing, but I'm honestly unsure what it is I am missing.