Speed phrenesics, scheduling, compactification.

ECS code, DAE, etc. Theoretical questions.
SFs are supposed to give damn fast responses, just like for a real user. Something or another.

Thus, when an SF is \{awakened\} \{interrupted\}, the scheduler has to do something snappier. The current guy has to be suspended, and the SF finished.

Let's say that at some point, time \( t_f \). Normally, the current command of the CW has accommodated the SF. It could be copied by brute force to ECS, or the SF Bracketed in a run. The CW is then restored, allowed to run. I know of two problems:

1) The CW may be in the middle of a program call, which is executing ECS code in some buffer. The SF may move out to buffer. It seems like the contents of the buffers have somehow to be preserved.

2) ECS may be all hank, cause compactification is in progress. The compactifier has to be told to cool it. I would allow some piece of time \( t_e \) to get itself straight. In this sense, I must...
be incremental.

3) The allocator isn't really reentrant.

The fight between $k_1$, $k_2$ is obvious. Two some things should be noted:

1) The SF is may may have to be recompiled.

2) The compactifier may be moving something gadgetulous. Either
   a) it is allocated to finish
   b) a mechanism for half-moving something had to be think up.

3) The SF better hadn't cause anything to be allocated (even destroyed is annoying).

4) if there's more than 1 SF, things get complicated fast

5) How long can I LOCK remain set?
Details of initiating a SF

A process may be fired up by:

1) getting an event
2) receiving an interrupt
3) being created.

For now, we consider only 1 (we’ll include 2+3 later if it fails out of (or we’re forced to)).

A) The interrupt code calls the event code calls the scheduler. The scheduler detects that it's a SF awakening.

B) The scheduler may have to determine what was interrupted:

1) user
2) system

so as to get into (out of) monitor mode correctly.