22 Feb '71

NEW STACK LOGIC

The manipulation of the call stack is being extensively revised. The most significant changes are:

1) When a subprocess does a system call, the p-counter in the stack will point at the XJ, not one beyond it;
2) A p-counter qualifier will indicate whether the subprocess was
   a) about to execute the inst at p-counter
   b) in the middle of the inst at p-counter (presumably an XJ)
   c) almost finished with the inst at p-counter "
3) A return action which will modify the p-counter qualifier of the previous stack entry as part of the action will be provided.
4) The interrupt inhibit bit will always be set when a new top of stack is formed, so that the current, running subprocess will automatically have interrupts inhibited. An operation to explicitly* set and clear the bit will also be provided.
5) The forced f-return and interrupt flags will disappear. I would like to move the interrupt inhibit bit from its present position if no one objects.

A complete description of the display stack operation and an "internals" specification for the new stack should be available soon.

Is there any enthusiasm for an action to display stack entries from some other process?

*Oh my God, a split infinitive!

THE LATEST WORD ON THE EXCITING STRUGGLE TO OVERCOME THE ELUSIVE BLOCK*GONE*FROM*FILE*IN*MAP/CHANGE*UNIQUE*NAME*OF*FILE PROBLEM (CONSIDERED IN CONJUNCTION WITH TURNING MAPS ON AND OFF)

Maps now have two counts on the compiled part and a new flag on the logical part

1) a local BADMAP count
2) a local COMPACTION count
3) a map on/off flag.

There is in addition a new flag on subprocesses, but it is very elusive. It says whether or not the subprocess is suffering from a pending map error.

To begin with the map on/off flag

1) An action to turn off the map of a specified subprocess will be produced in due course. It will decrement the map count on all file blocks used by the map and set the bit to off. (One gets an error for trying to turn off the map of a subprocess currently in the full path. Is that OK with everybody?)
2) doing anything that might cause a subprocess with its map turned off to swap in will cause an error, as discussed below.
3) The operation to turn the map back on will be fraught with all sorts of hazards stemming from missing blocks and files, but if one is lucky, it will find everything present that is necessary and increment the map count on all the relevant file blocks and turn the bit off.

When one changes unique names on a file, if the file has a block in a map, the map count on the block is cleard and a global BADMAP count is incremented. This leaves some map, somewhere, sitting around with one of its files ripped off. This may later lead to an error as discussed below.

I regret that I must also mention that some careless code may callously destroy a c-list that is the local c-list of some innocent subprocess, thereby causing said innocent subprocess grave emparrasment. (When the current swapper tries to bring in such a subprocess, it destroys the process!) But have no fear, relief is at hand.

How is one to see one's way out of this quagmire? Well, let's start with the hard-working swapping code, MAPOUT/MAPIN, which do the bulk of the systems swapping work. Here comes this subprocess to be swapped out/in. If the two local counts on the compiled map are up-to-date, and the map is on, the swap proceeds. But, if a count is off, further action is taken

0) if the map is off, an error is signaled to the caller (no swapping or compiling)
1) if the COMPACTION count is off, the map is recompiled
2) if the COMPACTION count is OK, but the BADMAP count is off, a check is made to see if all files in the logical map are still present; if so, the count is updated and the swap proceeds, but if not, the map is recompiled.

Whenever the map compiler encounters a missing file in a logical map, it zeros the map entry and proceeds with the compilation. It later exits with a signal if a file was gone. MAPOUT/MAPIN return this signal to whomever called them. The map is then swapped (with a possibly newly zeroed entry).

Now, if we're swapping, we're either swapping in or out, if you see what I mean. So, suppose we're swapping out and we get one of these funny errors from MAPOUT, what the hell do we do with it? Remember that the subprocess we just swapped out may not even be part of the full path currently, for reasons that are classified (the president knows best though, you may be sure). Well, we

1) ignore a mapoff error. Because you can only turn off the map of a subprocess that is out. This means that the mapoff condition was detected on swapin and the appropriate error generated as described later.
2) If a file was gone, since the entry was zeroed, the subprocess will swap back in later with nary a whimper, so we flag the subprocess at this point for a pending map error.

And that about covers swapout.

But what about swapping in? Here we can encounter three different hassles while just doing our job and minding our own buisness.

1) If the local c-list of the currnet subprocess has been ripped off, we generate the appropriate error right away.
2) If the logical map of one of the subprocesses in the full path is turned off, we generate the appropriate error.
3) If MAPIN reports that the logical map of one of the subprocesses in the full path has had a file ripped off, we also generate an error.
4) Last, if we see that one of the subprocesses that we are bringing in has a pending map error condition, we again generate the file-ripped-off error. The flag is turned off.

But what if they all happen at once? Only one gets reported, namely

1) the c-list error if it occurred
2) failing that, the type of map error occurring on the subprocess

closest to the current running subprocess is reported.

Well, I sure am glad to have that off my mind. Oh yes, I forgot to mention that when the map compiler encounters a missing block when it's compiling a logical map entry, it is still a DISASTER.

CONTROL OF CPU TIME

Two new features are being implemented for processes, a timer and an associated message mechanism. Time may be moved between the CPU time field of an allocation block and the timer of any of its owned proceses. When a process is swapped out, its timer is decremented by the time it just used, and if the result is negative, the process is descheduled; the negative residual sits in the timer.

The message mechanism, which is set by a separate system action, consists of an event channel and an event. When a process is descheduled, if the message mechanism is set, the swapper sends the event on the event channel (any errors, such as event channel gone or full, are ignored). If the message mechanism is not present, nothing further is done.

The operation which moves time into a process timer increments the current timer. If the process is descheduled and the timer goes positive, the process is rescheduled.

I heavily favor creating processes descheduled, but it is not too late to argue for an additional parameter on process creation to initialize the timer (that is the only alternative that I can see). A graceful phase-in will beprovided in any case.