2 April '70

INTERRUPTS

A) GENERal considerations, or, the conflict.

1) Three categories of interrupts are envisioned. In order of decreasing urgency, they are:

   a) The system wants to do something to the process because
      i) he's used all his money
      ii) people have to be swapped out to unjam ECS.

      These interrupts must be honored in a hurry.

   b) The user sends an interrupt from his TTY. (Two levels of urgency currently exist, CSP and BREAK.) The faster these interrupts take effect, the better the system looks.

   c) The user has given a friend a capability to interrupt him, but only wants it to strike under certain conditions. It doesn't matter if this type of interrupt never strikes.

2) There exist manipulations which cannot be terminated gracefully in mid-stream. Here the difference between interrupted and terminated makesxitsxif should be mentioned. Most things can be interrupted, provided that they are later allowed to finish without having been disturbed in any way in the interim. Examples of things which shouldn't be terminated at arbitrary times are

   a) the DISK SYSTEM, when it's twiddling pointers
   b) the DISK SYSTEM, when it's in an ungainly posture with respect to having something half-way swapped in or out
   c) the LINE COLLECTOR, when it's twiddling pointers
   d) a DATA BASE UPDATER, when it's updating

   The last one poses serious problems, because it is a manipulation which must be invokable by the user.

3) The orderly termination requirement conflicts with the semi-instantaneous response requirement. Any solution involves a compromise in that some small time interval must eventually be allowed for graceful completion of manipulations requiring graceful completion. Two radically different styles of solution have been discussed:

   a) Some sort of GLOBAL INTERRUPT INHIBIT BIT, or GIIB, which can be set locally to guarantee completion of critical operations. A bug which leaves this bit set indefinitely is intolerable, so that some mechanism of limiting the length of time that it is set must exist in the system. An implementation of this method is discussed in C below. Note that this scheme implies that allinterrupts are subject to some minimum delay whenever any critical manipulation is in progress.

   b) Making use of the current interrupt machinery, interrupts which must be honored fast are directed to an appropriately prestigious node of the subprocess tree (such as the root). The interrupt occurs immediately and then the SP fielding the interrupt has to decide what the hell to do with it. The bookeeping and implementation seem to be a nightmare, but this method is mentioned because it makes it possible to decide to interrupt something and then let it terminate later, giving potentially greater fesxibility and faster response than method a. A ghost of a suggestion as to implementation is given in D.

B) Current interrupt structure.

1) Subprocesses are arranged in a tree. Nodes above a given node are called its ancestors. A node is an ancestor of itself. Interrupts are directed to a particular subprocess, called the interrupt subprocess. An interrupt subprocess doesn't actually start execution until it becomes an ancestor of the subprocess currently executing, called the current subprocess. See fig 1.

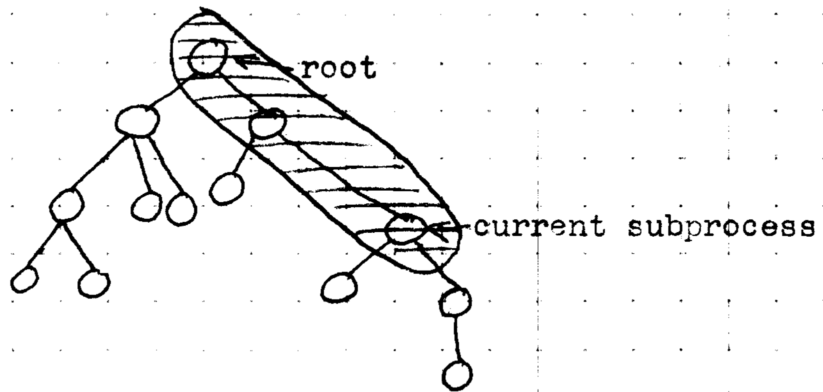

fig 1 - subprocess tree. interrupts directed to subprocess in the shaded area strike right away, modulo the IIB explained in 2; other interrupts wait.

2) Local interrupt inhibit bit (IIB). When an interrupt subprocess is fired up, an IIB is ~~set~~ automatically set which prevents the subprocess from recieving any further interrupts. The IIB goes away if the subprocess returns and doesn't have any effect if the subprocess has called another subprocess which is executing. The IIB may be set and reset by explicit system calls from within ~~t~~ the subprocess itself.

3) Interrupts arriving for ~~an interrupt subprocess~~ a pending interrupt subprocess are lost and have no effect; the first interrupt to arrive for a subprocess with the IIB set is remembered, subsequent ones are lost.

4) Howard justly observes that the tree structure for subprocesses serves a second function, namely, it determines how many nodes coexist in ~~storage~~ CM. An undesireable effect of this second use of the tree is that a subprocess which is logically an ancestor of some other sp may be put "off to the side", so as not to cramp it's (logical) descendents core. To salvage the interrupt logic, the ancestor must be split into a small piece, to intercept interrupts, and a main piece off to the side which is called by the small piece. This results in a proliferation of subprocesses.

C) Butler Lampson's (BCC's?) global interrupt inhibit with timer solution.

1) Basically, there is a GIIB which may be set and cleared by system calls. Associated with the GIIB is a real-time timer which is set to LIM whenever the GIIB is set. If the timer runs out while the GIIB is still set, error processing is initiated.

2) BCC allows a subprocess to set the GIIB even when it has already been set by a calling sp. So,
   a) the actual time that interrupts are locked out may by LIM*(depth of call stack), roughly.
   b) the GIIB has to be manipulated in the call stack, or some other stack.

3) The scheme makes it necessary for the system periodically to touch every process in the system (or every process on a list of processes with the GIIB set) to update the timer. When ~~the process is fired~~ timer runs out, the offending process must be fired up and error processing initiated. God knows what becomes of ~~pending~~ any interrupts pending on the process. Also, error processing has to be revamped to prevent undesireables from intercepting the error.

4) A big objection to the implementation of this scheme is the stack of timers - is it really necessary?

D) The magic, all-knowing subprocess solution. I can't get excited about really implementing this scheme, but a rough idea follows. It is a theoretically interesting solution, asit allows the possibility of "suspending" critical manipulations for later completion and avoids locking things up absolutely every time a manipulation deemed critical is being performed. (Consider the d case where the system has decided to destroy the process absolutely; it no longer seems too important to allow the LINE COLLECTOR to terminate gracefully)

1) Interrupts are handled as at present. Important interrupts are directed to sufficiently prestigious node of the tree.

2) The prestigious subprocess (PSP) is responsible for any idiosyncracies of his dependents. He decides what to do.

   a) If PSP decides to process the interrupt right away, there are no problems (unless, of course, he's wrong).

   b) If the PSP decides something critical might have to be wrapped up before processing the interrupt, he sets some kind of real-time timer and does a special call of the critical sp, warning it to tidy up. If the critical sp returns in time, fine. If not, we're in the same bag as when C's timer runs out.

E) Conclusions. As of this writing, we are short on conclusions. Everyone seems resigned to implementing some sort of GIIB with some sort of timer, but various people are still trying to conjure solutions simpler than C.

Also under discussion is the possible organization of the subprocess tree for the "typical user", vis-a-vis handling of various categories of interrupts. Nothing worth writing down has emerged from these discussions as yet. (People are still proposing radical alterations of the current interrupt structure. Boo-hiss.)