TO: R.C. Daley, G F McLaney, T VanVleck, K. Martin, S. Dunten, J. Saltzer

FROM: M. Thompson, P. Schicker

DATE: March 14, 1968

SUBJECT: System and Process Errors in Ring 0

Background:

Currently the standard way to terminate a bootload run in the event of an error or unexpected fault is to call or transfer to the segment Panic. Panic is an impure procedure which brings down one processor by executing a DIS instruction. Often this is not a useful way to terminate a multi-process or multi-processor run. The following notes outline a scheme to handle faults and errors in ring 0 after system initialization for a multi-process, multi processor environment, and suggest a way of using these pieces to clean-up fault and error handling during initialization.

I General Considerations

A. It is necessary to distinguish 3 different error conditions

1. system crash - brings down all processors
2. process_terminate - blocks one process
3. process_error - writes message and returns to command had.

B. During system initialization

1. it is desirable that all faults and interrupts be sent to the FIM and the II respectively as soon as possible
2. It is mandatory that all faults and interrupts be ~~divided~~ directed to the FIM and II before leaving ring 0.

3. It is desirable to have interrupt and fault initialization done in as few steps as possible and done in obvious places and in obvious ways.

II New (or almost new) modules and data required

A. pds$terminate_type

1. a new switch with two settings - system and process
   if set to system means to crash the system in case of error
   if set to process means to crash the process
2. It is used by terminate_proc, (panic)
3. Could be set and reset by fim and/or switch_stacks or maybe by the gate keeper to indicate when errors were fatal to the system and when only to the process.

B. Revised Panic program - (also referred to as terminate_proc)

0. Faults will not be directed to panic
1. It will retain an original entry like it has now (if necessary) to be used very early in the process's life before the first step of fault and ii init
2. has a new crash_switch; if this is off uses original method if on uses new method
3. New method: (note it is now a pure_procedure)

```
if pds$terminate_type = system
     call crash
```

if pds$terminate_type = process

call block

(it would be nice to have a hard_core ~~without~~ write procedure at this point to inform the user that he has just lost out)

C. Crash

1. Stops all processors (including it's own) by sending them system_trouble interrupts
2. has two entry points $dead, and $sleep when ~~at~~ entered sleep can be restarted by an execute fault
3. displays in the processor lights the usual fault information - i.e. A = fault

D. System_trouble_interrupt_handler

1. This must be built into ii since it may not be able to use temporary storage.
2. gets the interrupt frame prepared for a restart
3. masks against all interrupts and hangs on a ~~dis~~ DIS waiting for an execute fault, or a second ~~sys~~ SYS_trouble interrupt.

E. Ring 0 signal

1. has a fixed list of acceptable condition ~~n~~names and handlers and one ~~conformed~~ unclaimed_signal condition handler.
2. The three condition handlers are
   a. terminate_proc (panic) e.g. for illegal handler, some error returns; divide check etc.
   b. crash e.g. for MME's

c. crawl_out - e.g. some error returns (invalid_segno...) takes a return out of ring 0 which the gate keeper can recognize and cause the FIM to signal the condition in the new ring rather than return.

3. Before signal tries to crawl_out for any condition it will check that there is a ring to go to other than zero, if not it goes to terminate_proc.

III. Changes to system initialization

A. Bootstrap 2

1. leave interrupts going to bs 1
2. leave faults going to bs 1
   don't clear SDW for bs 1 ← one content will get used.

B. Fault and Interrupt Initialization step 1

This is done right after in MMCT_init so that we can read the clock for a timer runout fault or find out about all processors for a system_trouble_interrupt. Thus bs_2→initializer→Xray→RSW,→MMCT_INIT,→Fault_init_one

1. Fault_init
   a. initializes fim pointers (per bs 2)
   b. directs all faults to fim
   c. system_fault_dispatch goes to interim_2_segfault for segfault and process terminate for bound fault, gate faults, op.code not defined, ring 0 signal
   d. set pds$terminate_type = system

2. II_init

   a. initialize ii pointers

   b. send all interrupts except sys_trouble to ignore entry in ii

3. clear sdw of bs 1

C. II_init step 1a_

When turn_on_segment meter is called alarm-clock interrupts must be enabled.

D. Fault and Interrupt Initialization step 2

This is done after collection 3 is read in & initialized probably just before the call to init_proc

1. Fault_init

   a. reset fault_dispatch to go to real handlers

      gate

      bound fault

      op_code_not_defined

      ring 0 signal

2. II_init

   a. redirect interrupt vector to proper entries in the ii

Identification

crash

Purpose

Crash is called by the FIM (reason unexpected fault) or by panic alias terminate_proc (because of an unrecoverable situation). Any call to crash will bring the system to an ordinary stop.

Implementation

1. inhibit on, mask the calendar clock interrupts
2. set sys_trouble interrupts for all initialized processors and set their masks to allow only sys_trouble interrupts
3. redirect execute faults to <crash>/[sleep] the system is restartable.
4. inhibit off
5. wait for 1 msec to recieve the own sys_trouble interrupt
6. return (unless called at entry <crash>/[sleep] the system is restartable.

Restart:

1. inhibit, redirect execute faults to FIM
2. set sys_trouble for all initialized processors
3. DIS (will be interrupted by sys_trouble interrupt)

Identification

sys_trouble hander

Purpose

In order to bring a Multics System to a complete stop, the module "crash" will send sys_trouble interrupts to all processors associated with the System. Every processor will then stop on a DIS instruction in the sys_trouble handler. A further sys_trouble interrupt will restart the processor.

Implementation

1. The machine conditions of the interrupt ~~is~~ are stored in the current interrupt frame in the processor stack (PRDS). After this point, no other system data base may be used (there might be none available).
2. A switch is set to distinguish later restart interrupts.
3. The pointers in PRDS (stb_pointer, sreg_pointer, scu_pointer) are changed to point into a storage area inside the sys_trouble handler (nobody cares about this data)
4. The processor stops at a DIS instruction.
5. Upon restart (determined by examining the switch) the pointers in PRDS are restored (a copy of the original pointer in <prds> | [stb_pointer] was kept in <prds> | [stb_pointer]+6)
6. The restart switch is turned off
7. The processor mask is restored
8. The machine conditions are restored

Identification

signal in ring 0

Purpose

Signal in ring 0 is prelinked at sys init time. There is no signal vector in ring 0 but a list of conditions coded into signal. There are three possible actions:

1. shut down the system
2. terminate the process
3. crawl out of ring 0

Implementation

The condition name is compared with a list of condition names.

1. it is a "shut down" condition → call crash
2. it is a "terminate process" condition → call terminate_proc
3. none of the above
   a. is rtn_stk accessible? no→ call terminate_proc
   b. is rtn_stk initialized? no → call terminate_proc
   c. crawl out of ring 0

Crawl out

The ring 0 stack will be searched until the last ring cross flag is found; then execute the following sequence:

```
even
nop     *+2
rtcd    sp/20
oct     1253o77o3521
```

the rtcd instruction causes a wall crossing fault. FIM calls gate$out. After the usual ring switching procedure gatekeeper will detect the above mentioned sequence (1. cold instruction for rtcd, 2. is this a return from ring 0? 3. even instruction is nop, 4. address of nop instruction points at magic number) and modify the Fim of this special ~~constellation~~ condition. The FIM in turn will signal in whatever ring has called something in the hardcore ring.

RING 0 FAULTS AND ERRORS