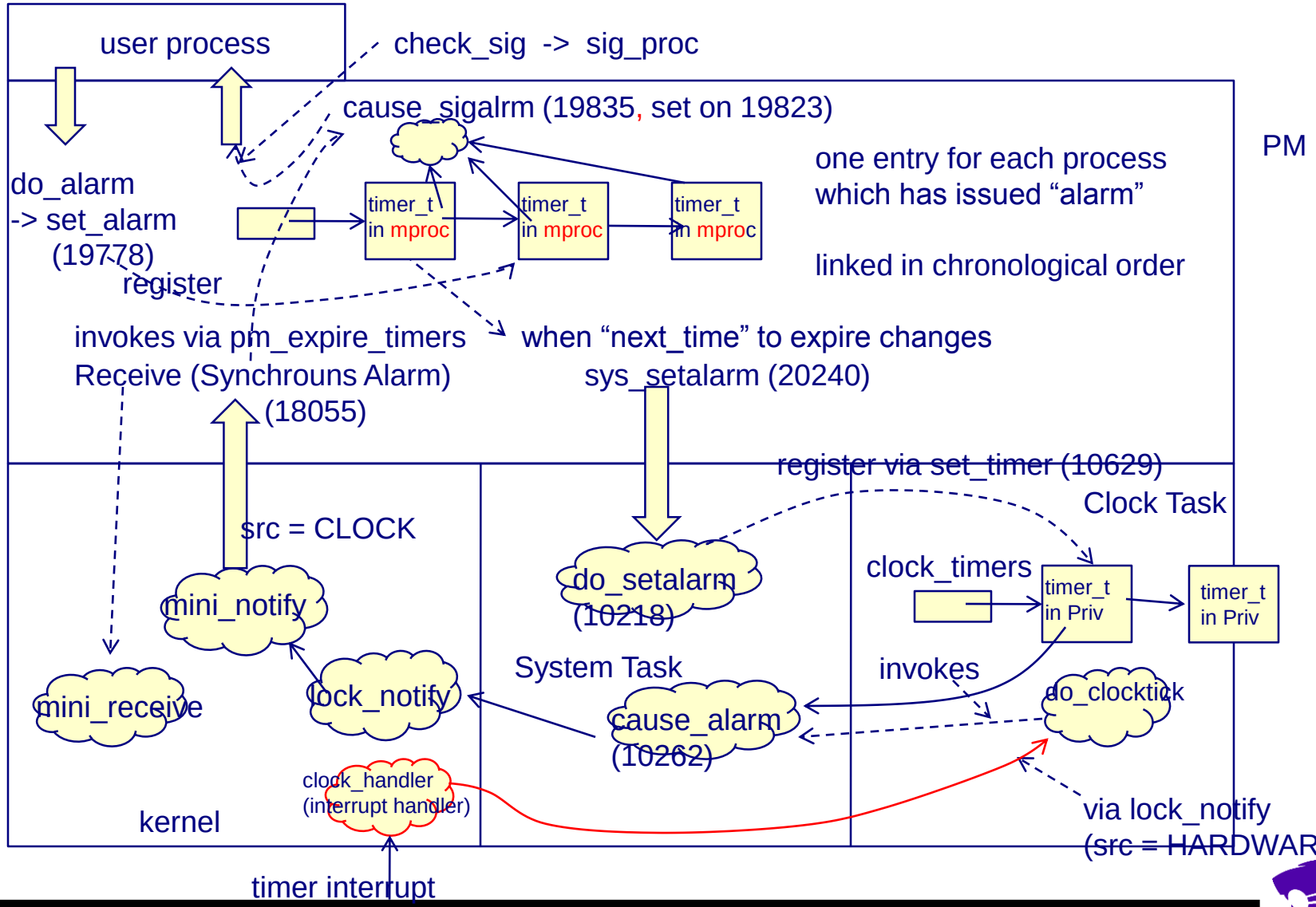
Three Ways to Deliver Signals

1. A process issues the kill system call
   - handled by do_kill (19689)
2. A clock timer expires
   - A notification message is from cause_alarm (10970)
   - call_nr = SYM_ALARM is sent to PM and pm_expire_timers (20256) handles this case
     - for each expired alarm, it calls cause_sigalarm (19835)
     - cause_sigalarm calls check_sig with SIGALRM
3. The kernel generates a signal (segmentation violation, SIGINT, SIGQUIT, SIGKILL etc) and sends a notification message to PM that contains pending signals (SYS_SIG)

- See the following define statements and recall notify messages (7458~7475)

```
03655 #define NOTIFY_MESSAGE          0x1000
03656 #define NOTIFY_FROM(p_nr)       (NOTIFY_MESSAGE | ((p_nr) + NR_TASKS))
03657  #define SYN_ALARM     NOTIFY_FROM(CLOCK)      /* synchronous alarm */
03658  #define SYS_SIG       NOTIFY_FROM(SYSTEM)     /* system signal */
```


Synchronous Alarm (revisited: see Chapter 6)



Handling Kernel Generated Signals

- Kernel calls cause_sig (9949)
  - e.g., see line 8059 in exception (8012)

Kernel (SYSTEM) side

cause_sig (dest_proc, sig_no) (9949) ⇨ add sig_no to dest_proc's set of pending signals
set dest_proc's SIGNALED, SIG_PENDING
and block it (STEP A)

↓ (9975)

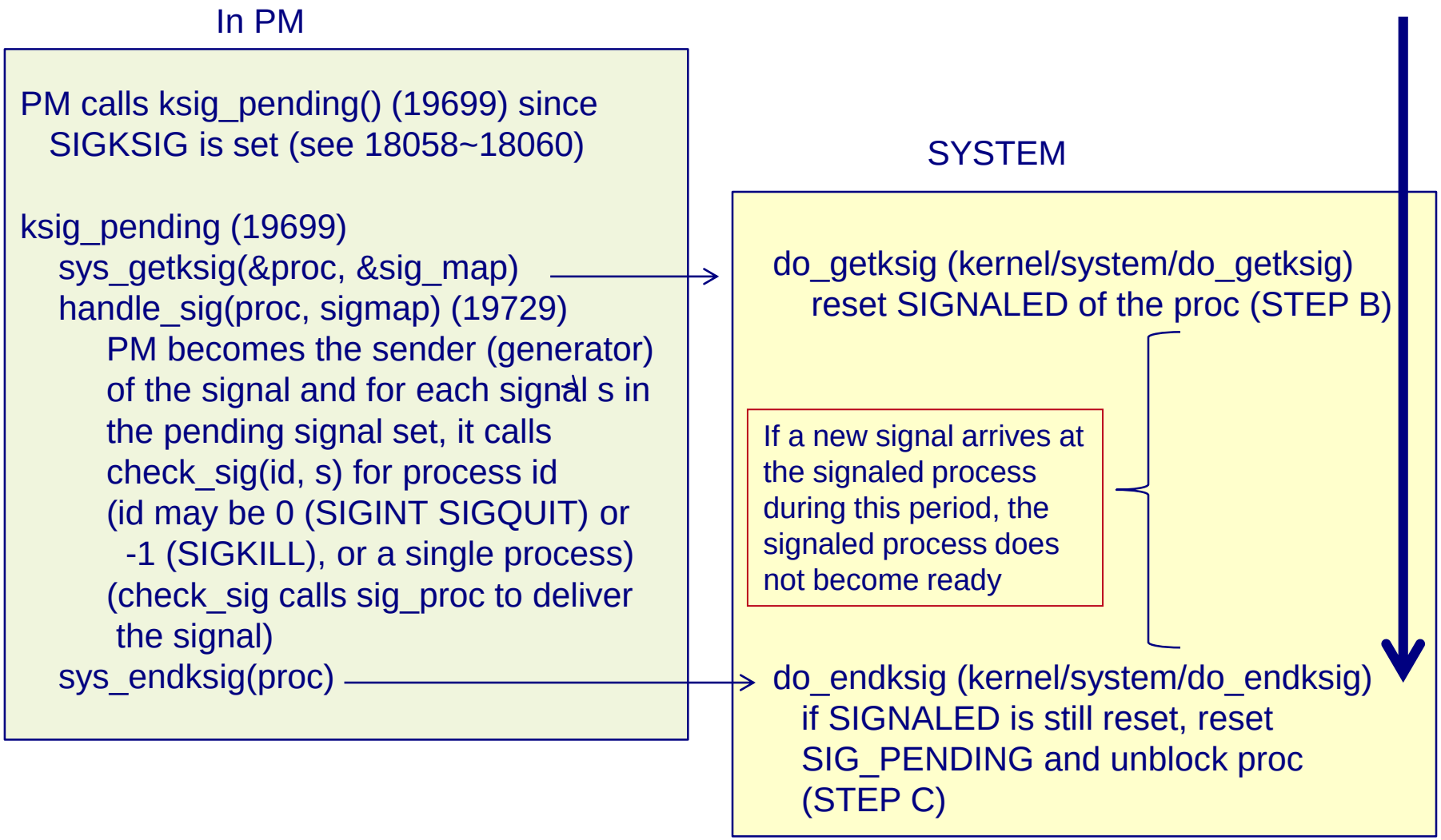
send_sig(PM, SIGKSIG) (9931)
SIGKSIG is added to PM's set of pending signals in the message

↓ (9943)

lock_notify(SYSTEM, PM) (7758)
message with call_nr = SYS_SIG is sent to PM

signaled process is blocked during this period

Handling Kernel Generated Signals (cont)



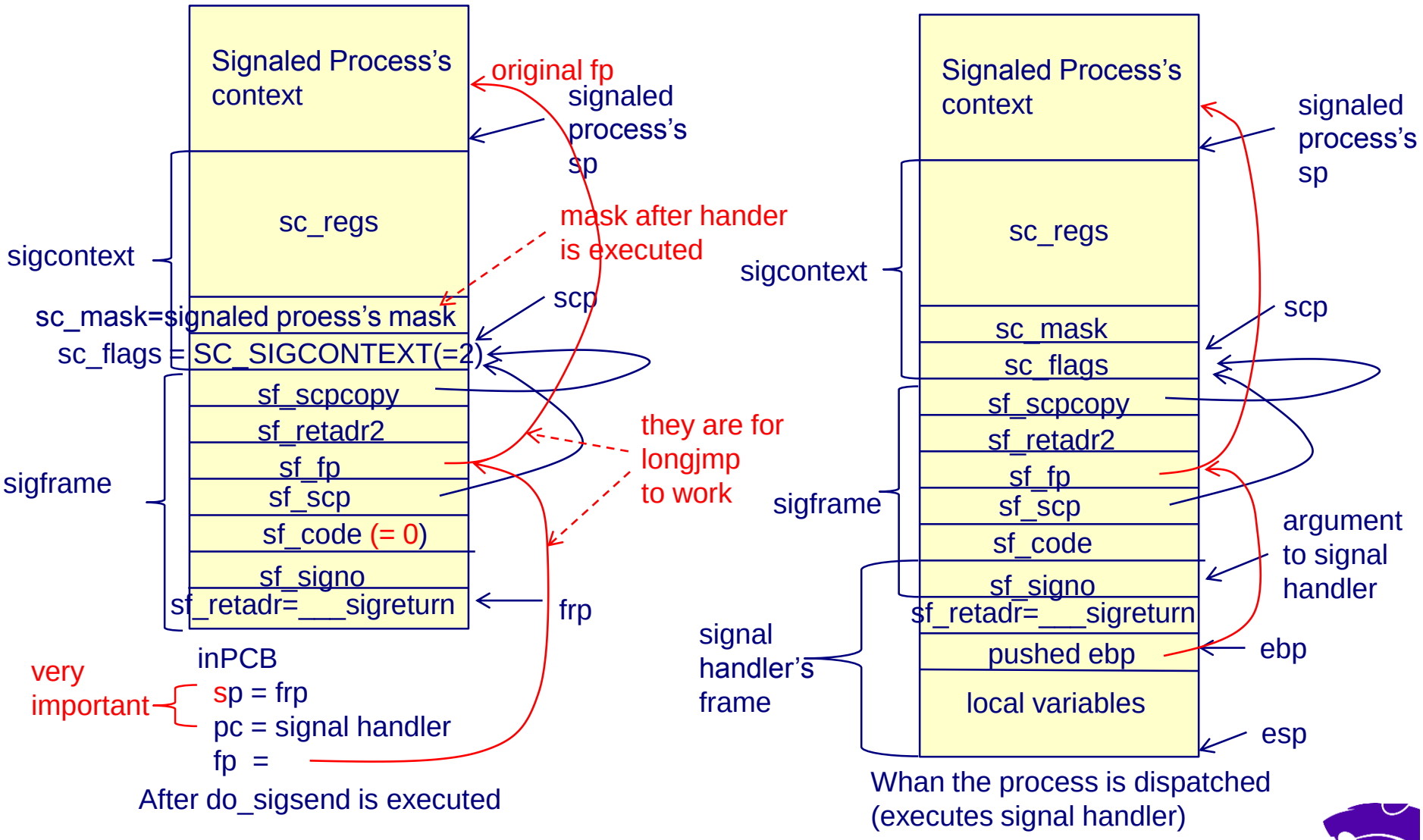
Interpretation of handling kernel generated signals

- Multiple kernel generated signals may be delivered and handled by a process at once
- In ksig_pending, PM sends all kernel generated signals to all processes that have pending kernel signals (see the while statement at 19714)
- The system wants to deliver a kernel generated signal to a process immediately
  - the process should not proceed when the kernel generates a signal to that process
  - Therefore, the kernel blocks the process to be signaled at STEP A
  - The process is unblocked when all the kernel generated signals have been sent (its stack and PCB have been modified) at STEP C

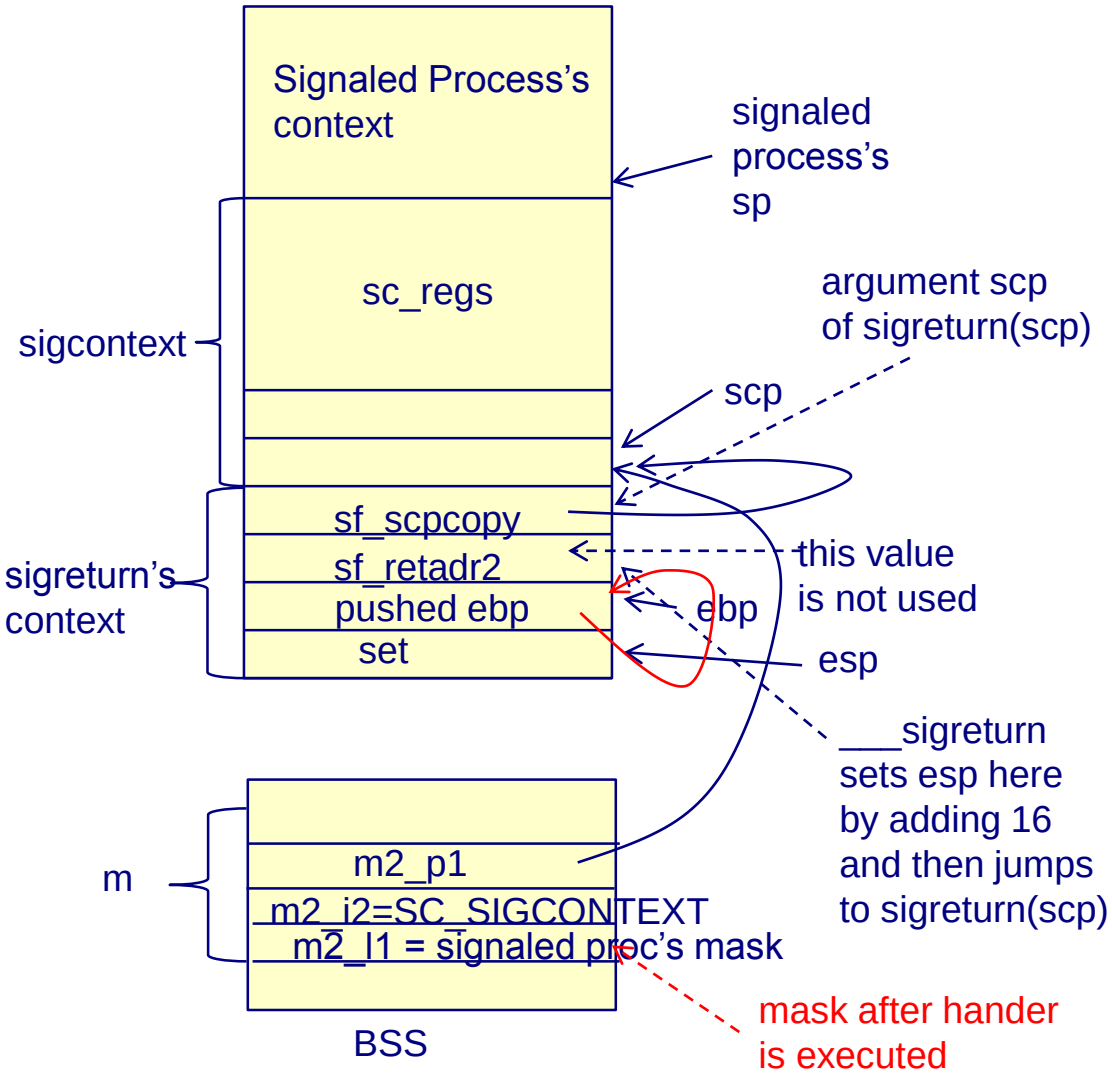
Interpretation of handling kernel generated signals (cont)

- Flag SIGNAL indicates that the kernel has generated a signal to a process (STEP A)
- When the system picks up a process to deliver pending kernel signals at STEP B, the process's SIGNAL flag is reset
- While PM is sending a signal to a process (changing its stack and PCB), the kernel may generate another signal to the process. Then, the process's SIGNAL flag is set again by STEP C
- Thus, in do_endksig(), the system unblocks the process only when its SIGNAL is still reset
  - If SIGNAL is set again, the process handles all (previous ones and the newly generated signals) signals (executes their signal handlers) next time when PM receives a message at 18052

Signal Processing

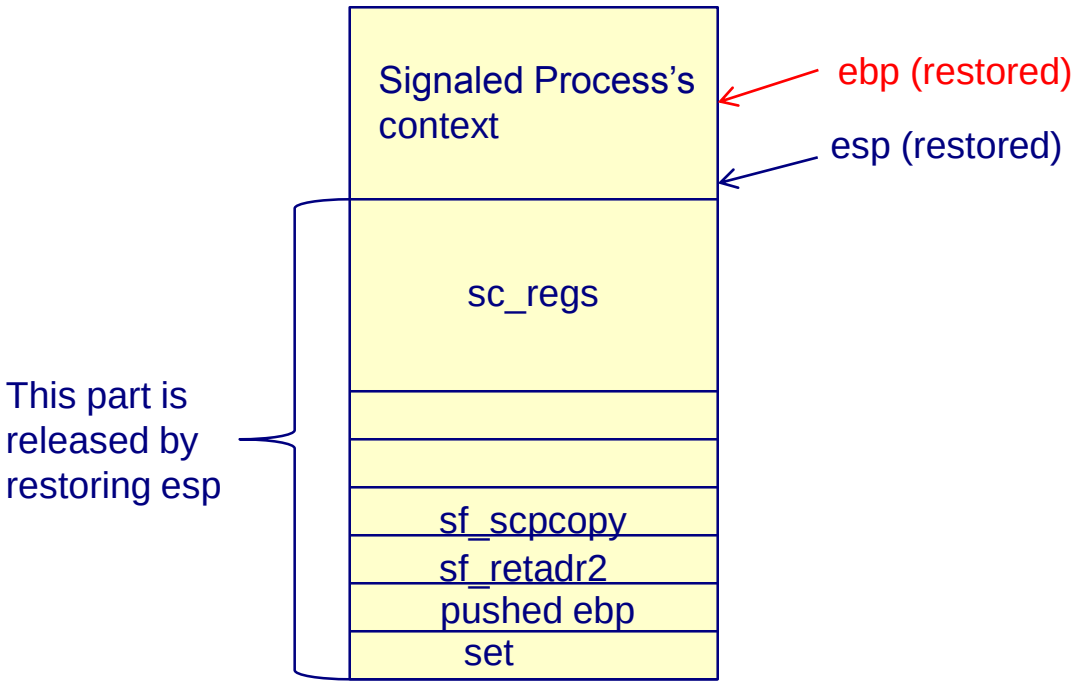


Signal Processing (cont)



- After the signal handler completes, it "returns" to ___sigreturn
- ___sigreturn adds 16 to esp and jumps to sigreturn(csp)
- sigreturn(scp) sets message m and passes the control to PM's do_sigreturn()
- PM's do_sigreturn() passes the control to Kernel's do_sigreturn with pointer to sigcontext [by casting it to (struct sigmsg *)]

Signal Processing (cont)



- Kernel's do_sigreturn copies sigcontext to its local variable sc
- Then, it restores sc's sc_regs to the signaled process's PCB (including eip (pc) and esp)
  - their values are set in 2 slides back
- When the process is dispatched, it resumes the execution from the point of signal
  - except for psw (psw holds the value at the end of the execution of the handler

check if this is semantically ok