5.7 Controlling return place after handler

- *Motivation*
  * Synchronous events, e.g. divide by zero, null pointer
  * Returning to the instruction causing error
  - implies infinite loop
  * Default: exit() or abort() in handler
  * Q? What if we don’t want to terminate process/thread?
- *Change place of return after handler*
  * Increment program counter (not portable)
  * Goto - missing stack management
  * siglongjmp => proper stack management
- *System calls: Siglongjmp(), sigsetjmp() (pp. 192)*
  * Like "goto" and "set label" but
  * Unravel function call stack properly!
  * Portable across POSIX.1 operating systems

5.7 Controlling return place after handler

- *Synopsis (pp. 192)*

```
#include <setjmp.h>
int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

- *Details of sigsetjmp()*
  - sigsetjmp() - set the "label" in "env"
  - argument 1 = stores <program counter, state of stack>
  - nonzero argument 2 => save signal_mask as well!
  - returns 0 if called directly
  - returns "val" when returning from siglongjmp()
    - like exit(v) -- wait(&status)
  - Call sigsetjmp() at the return point after handler
- *Details of siglongjmp()*
  - siglongjmp(env, val) - similar to "goto" label
  - argument 1 to restores <program counter, stack, mask>
  - argument 2 used to set return value for sigsetjmp()
  - Call inside the handler as the last step
  - Must have called sigsetjmp() before calling siglongjmp()

5.7 Controlling return place after handler

- *Example: Handling SIGSEGV, SIGFPE*

```
void handler( int sig ) {
        /* code fragment */
        siglongjmp(jmpbuf, 2);
}
void test() { int a = 0; a = a/ a ; /* SIGFPE */ }
int main() {
        /* install handler for SIGFPE, SIGSEGV */
        /* unmask SIGFPE, SIGSEGV */
        switch (status = sigsetjmp(jmpbuf, 1) ) {
                case -1: exit(0);
                case 0 : test() ;
                case 2 : /* code after return from handler */
  }
}
```

- *Ex. Program 5.2, pp. 192-3*
  - Q? Where do we return after int_handler()?
  - Q? What will the program print?
  - Q? Why use "jumpok" ?

5.7 Unwinding the Stack

- *objects on stack are destroyed*
  - local variables,
  - local class objects destructors are called
- *Program state (stack, mask) goes back to a previous state*
  - global data may have changed
  - existing objects may have changed
- *Issues*
  - What recovery is possible? Can data be saved?
  - What dynamic memory items need to be released?
  - Can intermediate changes to global data be undone?

5.7 Controlling return place after handler

- *Alternatives ways for unwinding stack*
- *ANSI Standard C Library*

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

  * POSIX : allows save/restore of signal mask
  * ANSI C : may be usable on non-POSIX OS

- *C++ try-catch blocks*

```
try {
        /* block monitored for exceptions */
        /* "throw" exception */
} catch(exceptiontype1 e) {
       /* exception handler code */
} catch(exceptiontype2 e) { /* handler code */
}
```

  * POSIX: allows save/restore of signal mask
  * C++ try-catch: not for asynchronous signals
  * C++ : more structured, easy to read/ debug
  * C++ : each block can have its own handler
  * recovery moves from 1 block to another block