Exceptions

CS 217

Handling Errors in C

- Return errors from a function

```
int foo( ... );

if (foo(...) == ERROR) {
   printf("error in function foo\n");
   exit(1);
}
```

- Problems
  - Client code may not check the error codes
    - `printf` returns the number of arguments successfully printed
    - Who checks that?
  - You may not have a chance to return an error code
    - Your code may have a divide-by-zero error

Handling Errors in C (cont'd)

- A global error flag `errno` to remember the last error of the system call
- Use `perror(const char *)`to print out the meaning of the error to `stderr`
  - Example

```
#include <stdio.h>

foo(...){
   ...
   perror("In function foo");
}
```

- Problem
  - Client may ignore the errors (forget about printing)

Exception Handling in Languages

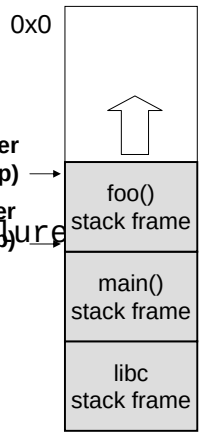
- Modern languages (Modula-2, Modula-3, C++, Java, etc) provide ways to handle exceptions
  - Programs can raise an exception
  - Catch the exception and handle it
- Try-Catch-Throw in C++

```
try {
   // code to be tried
   throw exception;
}
catch (type exception) {
   // code to be executed in case of exception
}
```

A C++ Example

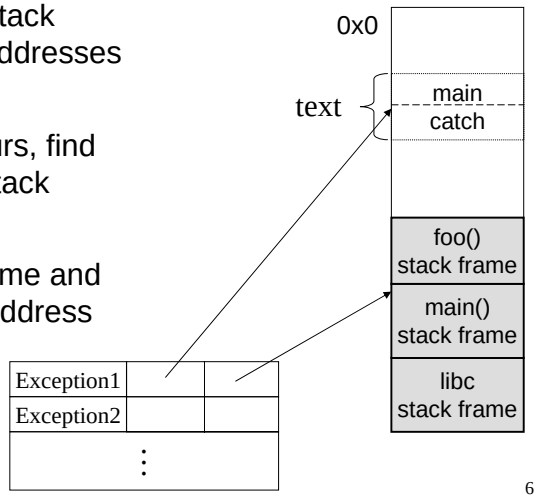
```
#include <iostream>
using namespace std;
foo(void) {
   char *buf;
   buf = new char[512];
   if( buf == 0 )
      throw "Memory allocation failure";
   ...
}

main(void) {
try {
      foo();
   }
```



Implementation Consideration

- For every “try-catch-throw”, register the scope (stack frame) and “catch” addresses in a data structure
- When a “throw” occurs, find the closest “catch” stack frame
- Unwind the stack frame and jump to the “catch” address



More Implementation Considerations

- Two kinds of exceptions
  - User defined
  - Predefined
- For predefined exceptions
  - For each “try-catch-throw”, register the scope (stack frame) and install a signal handler for finding the catch handler
  - When an exception occurs, OS invokes the handler which find the closest “catch” stack frame
  - Unwind the stack frame and jump to the “catch” address

Application’s “Context Switch”

- Problem
  - How do I write a program to interrupt a long printout and go back to the main processing loop?
- Calls

```
#include <setjmp.h>

int setjmp(jmp_buf env);
    /* save the stack environment */
void longjmp(jmp_buf env, int val);
    /* jump to the saved environment */
int sigsetjmp(sigjmp_buf env, int savemask);
    /* setjmp plus register, and signal mask */
void siglongjmp(sigjmp_buf env, int val);
    /* restore what sigsetjmp saved */
```

Example of setjmp and longjmp

```
#include <signal.h>
#include <setjmp.h>

static jmp_buf env;

void handler(int sig) {
   fprintf(stderr, “Interrupted\n”);
   longjmp(env, sig);
}
main(void)
{
   int returned;

   signal(SIGINT, handler);
   while ( ... )
      returned = setjmp(env);
           /* return 0 the 1st time, SIGINT 2nd time */
      /* main processing loop *
      ...
      long-print(bigfile);
      ;
}
```


Exception Handling in C?

- Can you use what we have learned to implement a simple exception handling mechanism in C?
  - Function based since we don’t have language support

Summary

- Exception handling
  - Use exception is a good way to handle errors and write a more robust program
- Two different ways to handle exceptions
  - User-defined exceptions
  - Predefined exceptions may need to install a signal handler to find the catch handler
  - Exception handling may need to deallocate memory of unwinded scope
- setjmp/longjmp in C can be used to switch context
  - Can be useful for implementing user-level threads (such as Java threads)