Subroutines and Control Abstraction

CSE 307 – Principles of Programming Languages

Stony Brook University

http://www.cs.stonybrook.edu/~cse307

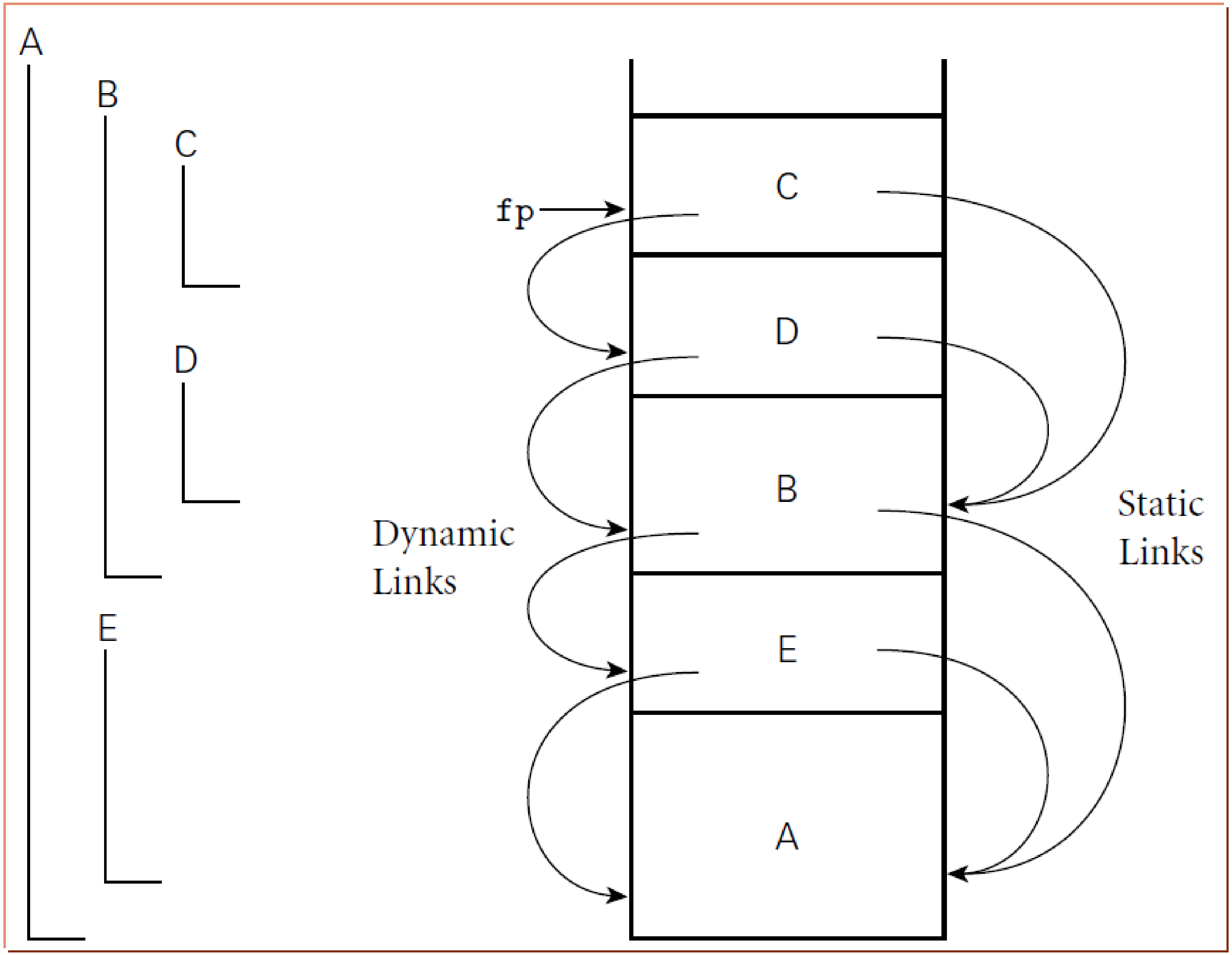
Subroutines

- Why use subroutines?
  - Give a name to a task.
  - We no longer care *how* the task is done.
- The *subroutine call* is an expression
  - Subroutines take arguments (in the formal parameters)
  - Values are placed into variables (actual parameters/arguments), and
  - A value is (usually) returned

Review Of Memory Layout

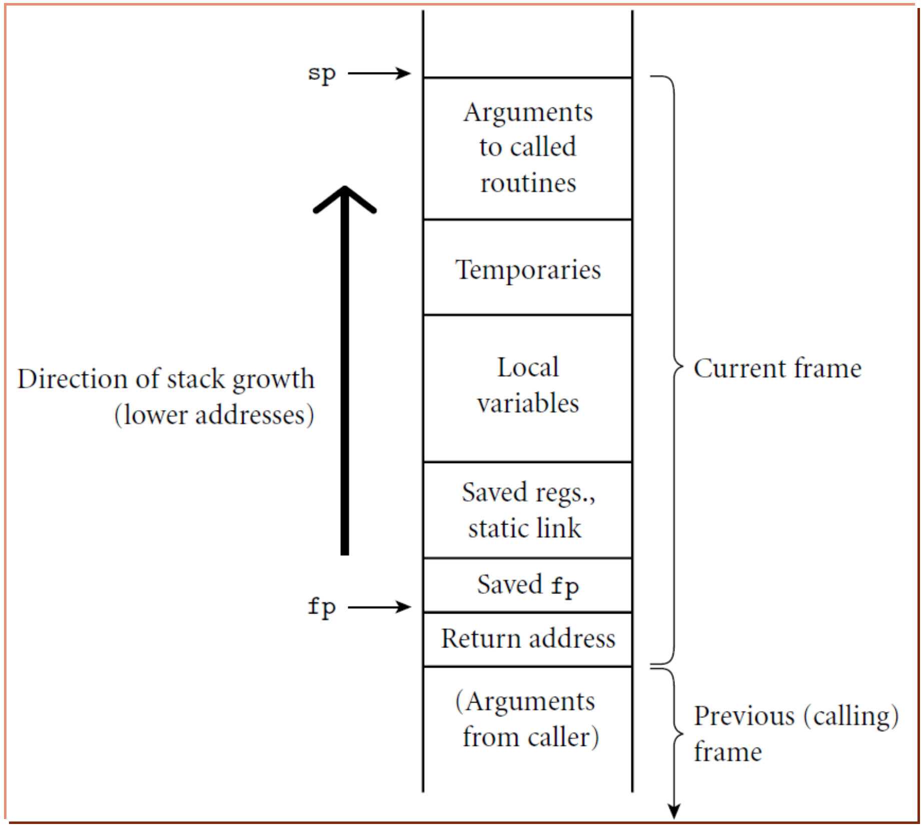
- Allocation strategies:
  - Static
    - Code
    - Globals
    - Explicit constants (including strings, sets, other aggregates)
    - Small scalars may be stored in the instructions themselves
  - Stack
    - parameters
    - local variables
    - temporaries
    - bookkeeping information
  - Heap
    - dynamic allocation

Review Of Stack Layout



Review Of Stack Layout

- Contents of a stack frame:
  - bookkeeping
    - return Program Counter
    - saved registers
    - line number
    - static link
  - arguments and returns
  - local variables
  - temporaries



Calling Sequences

- Maintenance of stack is responsibility of *calling sequence* and subroutines *prolog* and *epilog*
  - Tasks that must be accomplished on the way into a subroutine include passing parameters, saving the return address, changing the program counter, changing the stack pointer to allocate space, saving registers (including the frame pointer) that contain important values and that may be overwritten by the callee, changing the frame pointer to refer to the new frame, and executing initialization code for any objects in the new frame that require it.
  - Tasks that must be accomplished on the way out include passing return parameters or function values, executing finalization code for any local objects that require it, deallocating the stack frame (restoring the stack pointer), restoring other saved registers (including the frame pointer), and restoring the program counter.
- space is saved by putting as much in the prolog and epilog as possible
- time may be saved by putting stuff in the caller instead, where more information may be known

Calling Sequences

- The trickiest division-of-labor issue pertains to saving registers
  - The ideal approach is to save precisely those registers that are both in use in the caller and needed for other purposes in the callee
  - Common strategy is to divide registers into ***caller-saves*** and ***callee-saves*** sets
  - Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time
- Some storage layouts use a separate ***arguments pointer***

Calling Sequences (LLVM on ARM)

- Caller:
  - saves into the "*local variable and temporaries*" area any caller-saves registers whose values are still needed
  - puts up to **4** small arguments into registers **`r0-r3`**
  - puts the rest of the arguments into the argument build area at the top of the current frame
  - puts return address into register **`lr`**, jumps to target address, and (optionally) changes instruction set coding

Calling Sequences (LLVM on ARM)

- In prolog, Callee
  - pushes necessary registers onto stack
  - initializes frame pointer by adding small constant to the **`sp`** placing result in **`r7`**
  - subtracts from **`sp`** to make space for local variables, temporaries, and arg. build area at top of stack
- In epilog, Callee
  - puts return value into **`r0-r3`** or memory (as appropriate)
  - subtracts small constant from **`r7`**, puts result in **`sp`** (effectively deallocates most of frame)
  - pops saved registers from stack, pc takes place of **`lr`** from prologue (branches to caller as side effect)

Calling Sequences (LLVM on ARM)

- After call, Caller
  - moves **return value** to wherever it's needed
  - restores caller-saves registers lazily over time, as their values are needed
- All arguments have space in the stack, whether passed in registers or not
- The subroutine just begins with some of the arguments already cached in registers, and 'stale' values in memory

- **Optimizing compilers keep things in registers whenever possible, flushing to memory only when they run out of registers, or when code may attempt to access the data through a pointer or from an inner scope**

Calling Sequences (LLVM on ARM)

- Many parts of the calling sequence, prologue, and/or epilogue can be omitted in common cases
  - leaving things out saves time
  - simple leaf routines don't use the stack - don't even use memory – and are exceptionally fast

Parameter Passing

- Modes of passing parameters:
  - Call by value: make a copy of the parameter.
  - Call by reference (aliasing): allows the function to change the parameter
    - out-parameters
  - Call by sharing: requires parameter to be a reference itself.
    - Makes copy of reference that initially refers to the same object.
    - E.g., Python, Java Objects.

Parameter Passing

```
def f(a):
   a += 1
x = 0
f(x)
print(x)
```

- value: 0
- reference: 1
- sharing: 0

Parameter Passing

```
def f(a):
    a.foo = 1
x = object()
x.foo = 0
f(x)
print x.foo
```

- value: 0
- reference: 1
- sharing: 1

Parameter Passing

```
z = object()
z.foo = 1
def f(a):
    a = z
x = object()
x.foo = 0
f(x)
print x.foo
```

- value: 0
- reference: 1
- sharing: 0

Parameter Passing

- Call-by-value:
  - Can be expensive to implement (e.g., copying large objects).
  - Can't change a parameter, except by returning a new copy.
- Call-by-reference:
  - Out-parameters (i.e., the procedure returns values through its parameters).
  - Good: More flexibility.
  - Bad: Can be confusing when arguments change.

Parameter Passing

- C/C++ functions
  - parameters passed by value (C)
  - parameters passed by reference can be simulated with pointers (C)

```
void proc(int* x,int y){*x = *x+y }
  …
proc(&a,b);
```

  - or directly passed by reference (C++)

```
void proc(int& x, int y) {x = x + y }
proc(a,b);
```

Parameter Passing

- Call-by-sharing:
  - No copying of large objects.
  - No implicit out parameters.
    - Can change objects, but not arguments.

Parameter Passing

- Other fun tricks with parameters:
  - ***Named parameters (pass-by-name)***: the values are passed by ***associating*** each one with a ***parameter name***. E.g.,in Objective-C:

```
[window addNewControlWithTitle:@"Title"
        xPosition:20
        yPosition:50
            width:100
           height:50
       drawingNow:YES];
```

Parameter Passing

- ***Default parameters***: default values are provided to the function
  - C++ example:

```
void PrintValues(int nValue1, int nValue2=10){
    using namespace std;
    cout << "1st value: " << nValue1 << endl;
    cout << "2nd value: " << nValue2 << endl;
}
int main(){
    PrintValues(1);
    // nValue2 will use default parameter of 10

    PrintValues(3, 4);
    // override default value for nValue2
}
```

Parameter Passing

- ***Variadic functions***: functions of **indefinite** arities
  - C, Objective-C and C++:

```
double average(int count, ...){
    va_list ap;
    int j;
    double tot = 0;
    //Requires the last fixed parameter (to get the address)
    va_start(ap, count);
    for(j=0; j<count; j++)
        tot+=va_arg(ap, double);
        //Requires the type to cast to
        // Also increments ap to the next argument.
    va_end(ap);
    return tot/count;
}
```

Parameter Passing

- ***Pass-by-name in ALGOL 60***:
  - the body of a function is interpreted at call time after textually substituting the actual parameters into the function body.
  - In this sense the evaluation method is similar to that of C preprocessor macros.
  - By substituting the actual parameters into the function body, the function body can both read and write the given parameters. In this sense the evaluation method is similar to pass-by-reference.
    - The difference is that since with pass-by-name the parameter is *evaluated* inside the function, a parameter such as a[i] depends on the current value of i inside the function, rather than referring to the value of a[i] before the function was called.

Parameter Passing

- **Pass-By-Name Security Problem**:

```
procedure swap (a, b);
     integer a, b, temp;
     begin
       temp := a;
       a := b;
       b:= temp
     end;
Call swap(i, x[i]); // What happens?
      temp := i;  i := x[i]; x[i] := temp
```

Before call:	i = 2	x[2] = 5	
After call:	i = 5		x[5] = 5

Swap doesn't work!

Parameter Passing

- ***<u>Pass by Value-Returned (or value-result)</u>*: pass a value-returned parameter by address (just like pass by reference parameters), but, upon entry, the procedure makes a temporary copy of this parameter and uses the copy while the procedure is executing.
  - When the procedure finishes, it copies the temporary copy back to the original parameter.
  - In some instances, pass by value-returned is more efficient than pass by reference, in others it is less efficient:
    - If a procedure only references the parameter a couple of times, copying the parameter's data is expensive.
    - If the procedure uses this parameter often, the procedure amortizes the fixed cost of copying the data over many inexpensive accesses to the local copy.

Parameter Passing

- ***<u>Pass by Result</u>***: almost identical to pass by value-returned: the procedure uses a local copy of the variable and then stores the result through the pointer when returning.
  - The difference between pass by value-returned and pass by result is that when passing parameters by result you do not copy the data upon entering the procedure.
    - Pass by result parameters are for returning values, not passing data to the procedure.
    - Therefore, pass by result is slightly more efficient than pass by value-returned since you save the cost of copying the data into the local variable.

Parameter Passing

Parameter mode	Representative languages	Implementation mechanism	Permissible operations	Change to actual?	Alias?
value	C/C++, Pascal, Java/C# (value types)	value	read, write	no	no
`in`, `const`	Ada, C/C++, Modula-3	value or reference	read only	no	maybe
`out`	Ada	value or reference	write only	yes	maybe
value/result	Algol W	value	read, write	yes	no
`var`, `ref`	Fortran, Pascal, C++	reference	read, write	yes	yes
sharing	Lisp/Scheme, ML, Java/C# (reference types)	value or reference	read, write	yes	yes
r-value ref	C++11	reference	read, write	yes*	no*
`in out`	Ada, Swift	value or reference	read, write	yes	maybe
name	Algol 60, Simula	closure (thunk)	read, write	yes	yes
need	Haskell, R	closure (thunk) with memoization	read, write†	yes†	yes†

Figure 9.3 **Parameter-passing modes.** Column 1 indicates common names for modes. Column 2 indicates prominent languages that use the modes, or that introduced them. Column 3 indicates implementation via passing of values, references, or closures. Column 4 indicates whether the callee can read or write the formal parameter. Column 5 indicates whether changes to the formal parameter affect the actual parameter. Column 6 indicates whether changes to the formal or actual parameter, during the execution of the subroutine, may be visible through the other. *Behavior is undefined if the program attempts to use an r-value argument after the call. †Changes to arguments passed by need in R will happen only on the first use; changes in Haskell are not permitted.

Returning from a Function

- Different ways of returning a value from a function.
  - Return statement
  - Assigning to the function name (Pascal, Fortran, Algol)
    - This interacts poorly w/ scoping and recursion
  - Special return location
    - Eiffel calls it **`Result`**
      - Means we don't have to allocate a variable to store the result in

Generic Subroutines and Modules

- Generic modules or classes are particularly valuable for creating containers: data abstractions that hold a collection of objects
  - When defining a function, we don't need to give all the types
  - When we invoke the class or function we specify the type: *parametric polymorphism*
- Generic subroutines (methods) are needed in generic modules (classes), and may also be useful in their own right (see Java **`static`** generic methods)

Generic Subroutines and Modules

- Generic programming in programming languages:
  - One approach is *implicit parametric polymorphism*:
    - Dynamic typing.
      - Just try running the code.
      - No checking at compile time - not type safe.
    - **Python approach**.
  - An alternative is to have a function that has parameterized types: *explicit parametric polymorphism*
    - Generic classes and methods
    - Can be static typed checked
    - **Java approach**

```
static boolean <T> allEqual(T a, T b, T c) {
     return a.equals(b) && b.equals(c);
}
```

Generic Subroutines and Modules

- Implementation approaches:
  - C++:
    - generates new code for each type:
      - linker can help with that
      - allows specialization
      - can make the code bigger
      - can use types in the function: **`new T();`**
      - Templates can cause horrible error messages
  - Java
    - *type erasure*: replace all type parameters in generic types with their bounds
      - Only one instance of the code at run time
      - Can't do operations involving the type

Generic Subroutines and Modules

- Generics are better than macros:
  - E.g., take the macro:

```
#define min(a, b) (a < b) ? a : b
```

  - Problem: `min(a++, b++)`
    - Variables **a++** or **b++** evaluated more than once
  - C++ generic:

```
template <class T>
T min(T a, T b) {
  return (a < b ) ? a : b;
}
```

    - Far fewer problems: variables evaluated only once.

Exception Types

- What is an exception?
  - a hardware-detected run-time error or unusual condition detected by software
- Examples
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions, not necessarily errors

Exception Handling

- What is an exception handler?
  - code executed when exception occurs
  - may need a different handler for each type of exception
- Why design in exception handling facilities?
  - allow user to explicitly handle errors in a uniform manner

Coroutines

- Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that transfer control to each other explicitly, by name
  - Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack

Coroutines

- As a simple application, consider a "screen-saver" program, which paints a mostly black picture on the screen of an inactive workstation, and which keeps the picture moving (to avoid phosphor or liquid-crystal "burn-in"), and also performs "sanity checks" on the file system in the background, looking for corrupted files
  - We could write a loop which does screen update and check in one block, but this mixes tasks
  - Better: use coroutines

```
coroutine check file system
  for all files …
coroutine update screen
  loop
        update screen
```

Coroutines

- Can implement threads
- Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack: their subroutine calls and returns, taken as a whole, do not occur in last-in-first-out order.
  - If each coroutine is declared at the outermost level of lexical nesting, then their stacks are entirely disjoint
    - The simplest solution is to give each coroutine a fixed amount of statically allocated stack space
    - But they may still use static links for scoping

Coroutines

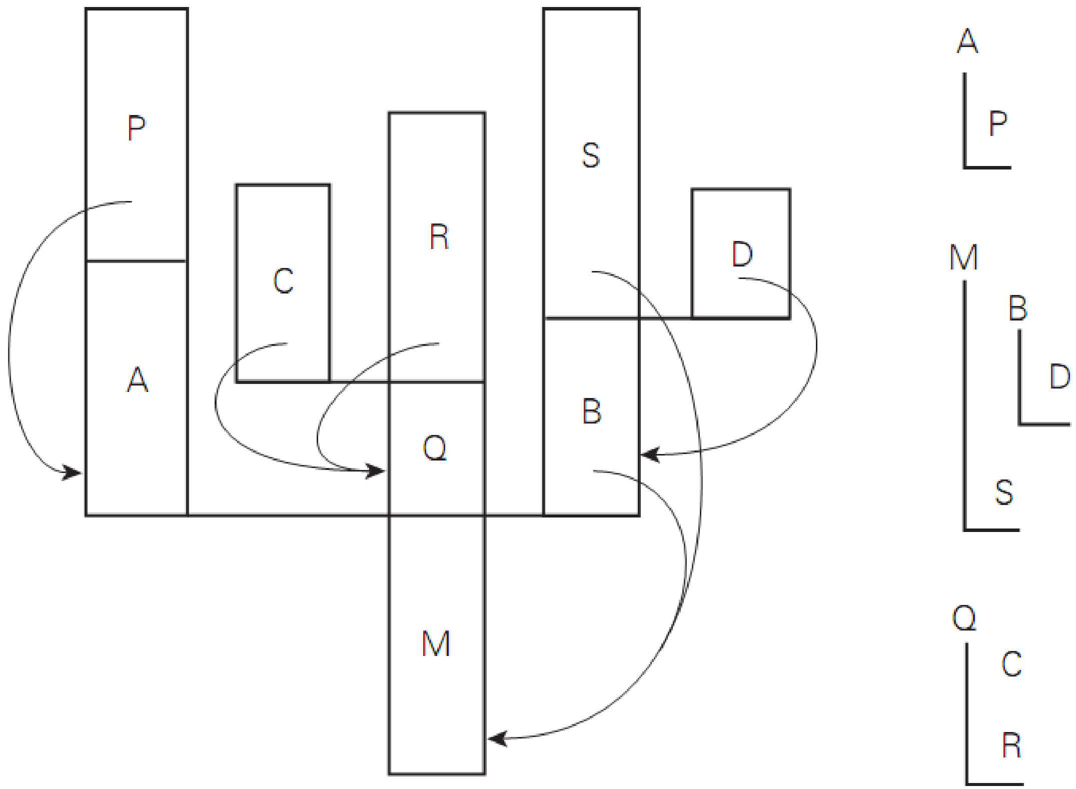


Figure 8.6 A cactus stack. Each branch to the side represents the creation of a coroutine (A, B, C, and D). The static nesting of blocks is shown at right. Static links are shown with arrows. Dynamic links are indicated simply by vertical arrangement: each routine has called the one above it. (Coroutine B, for example, was created by the main program, M. B in turn called subroutine S and created coroutine D.)

Event types

- An event is something to which a running program (a process) needs to respond, but which occurs outside the program, at an unpredictable time.
  - The most common events are inputs to a graphical user interface (GUI) system: keystrokes, mouse motions, button clicks.
  - They may also be network operations or other asynchronous I/O activity: the arrival of a message, the completion of a previously requested disk operation
- A handler—a special subroutine— is invoked when a given event occurs.
- Thread-Based Handlers:
  - In modern programming languages and run-time systems, events are often handled by a separate thread of control, rather than by spontaneous subroutine calls
    - With a separate handler thread, input can again be synchronous: the handler thread makes a system call to request the next event, and waits for it to occur.
    - Meanwhile, the main program continues to execute.
    - Many contemporary GUI systems are thread-based.
      - most use anonymous inner classes for handlers

Event types

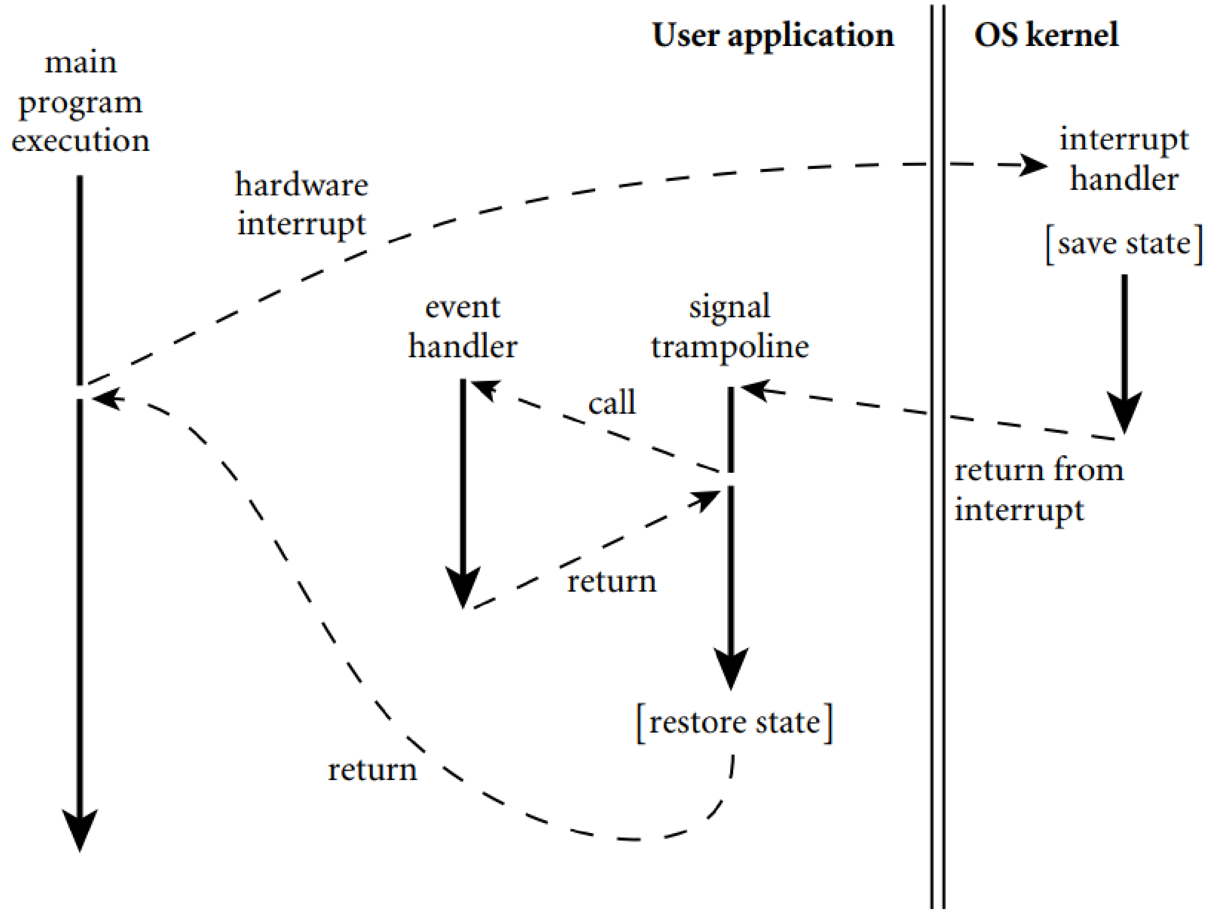


Figure 8.7 Signal delivery through a trampoline. When an interrupt occurs (or when another process performs an operation that should appear as an event), the main program may be at an arbitrary place in its code. The kernel saves state and invokes a *trampoline* routine that in turn calls the event handler through the normal calling sequence. After the event handler returns, the trampoline restores the saved state and returns to where the main program left off.

Summary

- Functional Abstraction:
  - Functions help us abstract the code:
    - by being able to give parts of the program meaningful name
    - by creating scopes in which data and control flow is controlled.
  - 3 main calling conventions.
    - Pass by value.
    - Pass by reference.
    - Pass by sharing.
  - Generics
  - Exceptions
  - Coroutines
  - Events