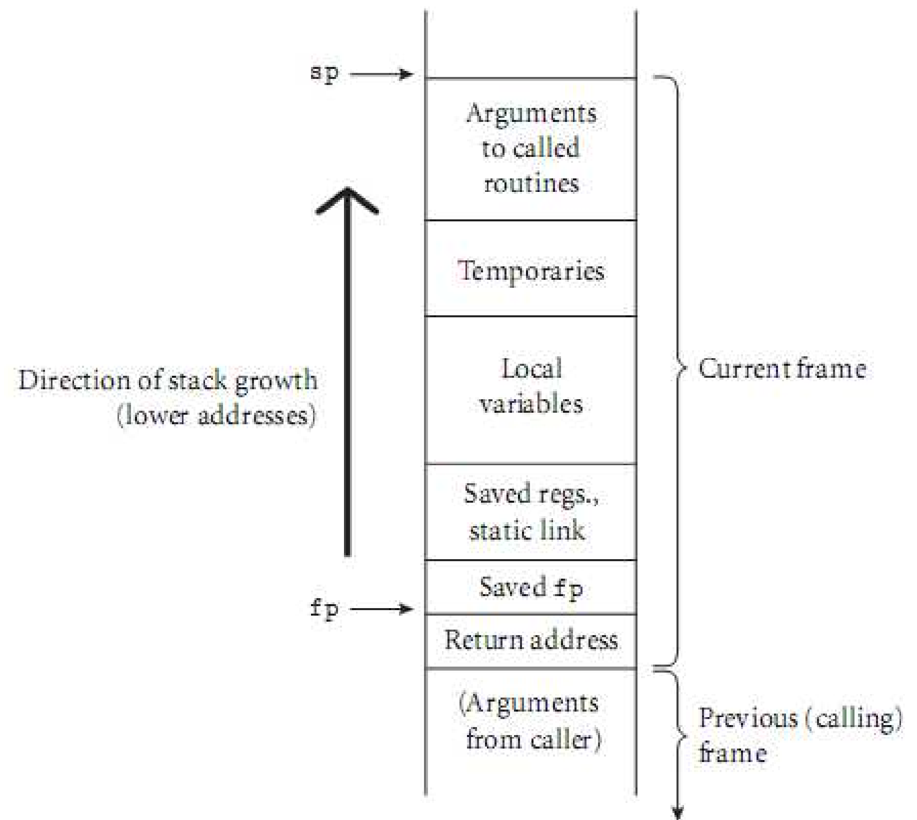
남 범 석

Review: Stack Frame

- Contents of a stack frame
  - Bookkeeping
    - Return PC (dynamic link)
    - Saved registers
    - Line number
    - Static link
  - Arguments and returns
  - Local variables
  - Temporaries



Parameter Passing

- Three basic implementations for parameter passing
  - Call-by-value
  - Call-by-reference
  - Call-by-closure (subroutine + environment)

- Variations
  - Call-by-result – copying back when return
  - Call-by-value/result – copying and copying back
  - Call-by-sharing – copying in reference model
  - Call-by-name – substitute as macros
  - Read-only – prevent modification
  - Named parameter – specify corresponding formal parameters

Parameter Passing in C/C++

- 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 in Ada

- Ada goes for semantics: who can do what
  - in: callee reads only
  - out: callee writes and can then read (formal not initialized); actual modified
  - in out: callee reads and writes; actual modified

Parameter Passing in Others

- Language with a reference model of variables (Lisp, Clu)
  - Pass by reference (*sharing*) is the obvious approach

- Fortran always uses call-by-reference (only option)
  - If you pass a constant or expression, the compiler creates a temporary location to hold the value and pass its reference

- Call-by-name is an old Algol technique
  - Think of it as call by textual substitution
    (procedure with all name parameters works like macro)
  - A way to mimic macros for assembly language programmers

Parameter Passing Modes

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
`in out`	Ada	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*

Exceptions

- 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 users to explicitly handle errors in a uniform manner
  - Allow users to handle errors without having to check these conditions before
  - Explicitly in the program everywhere they might occur

Events

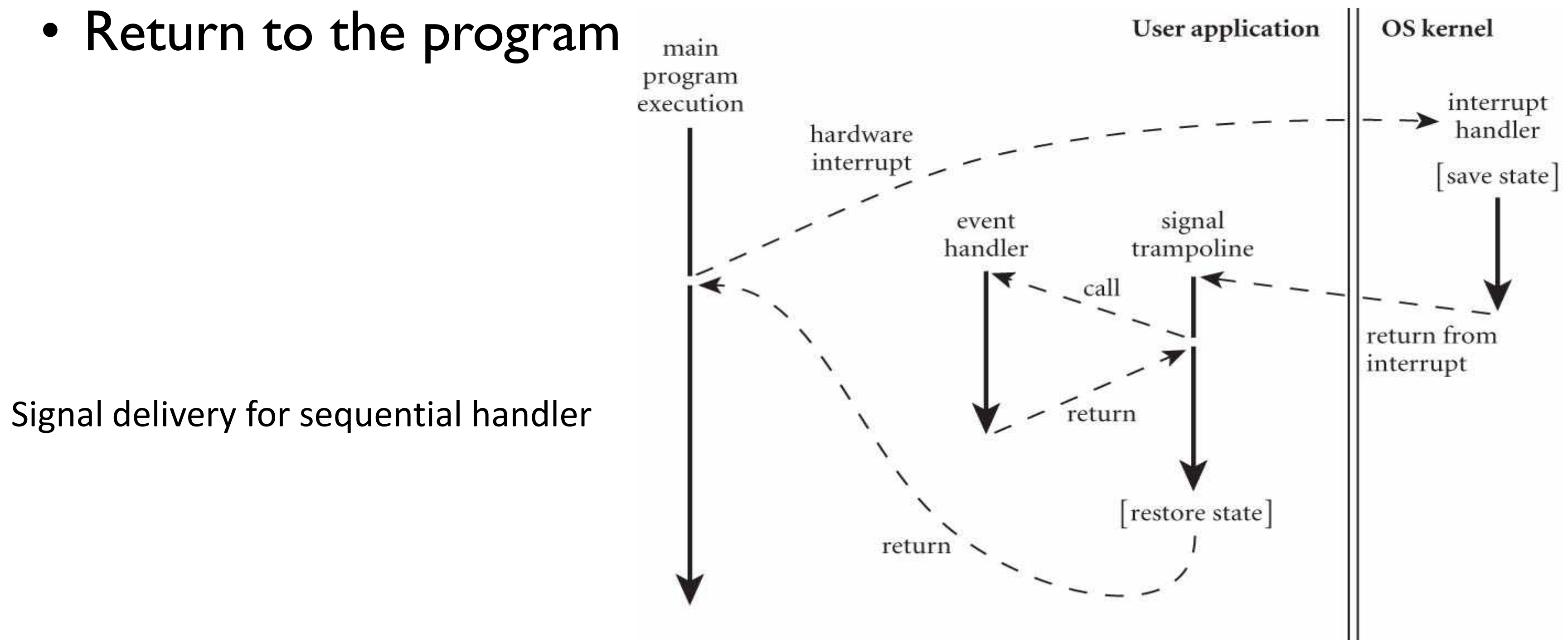
- What is an event?
  - Running programs need to respond to the events
  - Events occurs outside the programs at unpredictable times

- Examples
  - Inputs from users (keyboard, mouse motion/click)
  - I/O from network, disk

- Event handlers
  - A special callback function
  - A dedicated thread to handle

Event Handlers

- Sequential handlers
  - Works in an asynchronous way
    - Register handler and return
  - Use a hardware interrupt mechanism
  - OS calls handler routines
  - Return to the program



Signal delivery for sequential handler

Event Handlers

- Thread-based handlers
  - Separate thread control thread
  - Works in a synchronous way
    - A dedicated thread is called
    - Makes a system call for the next event
    - Waits for it to occur

Coroutines

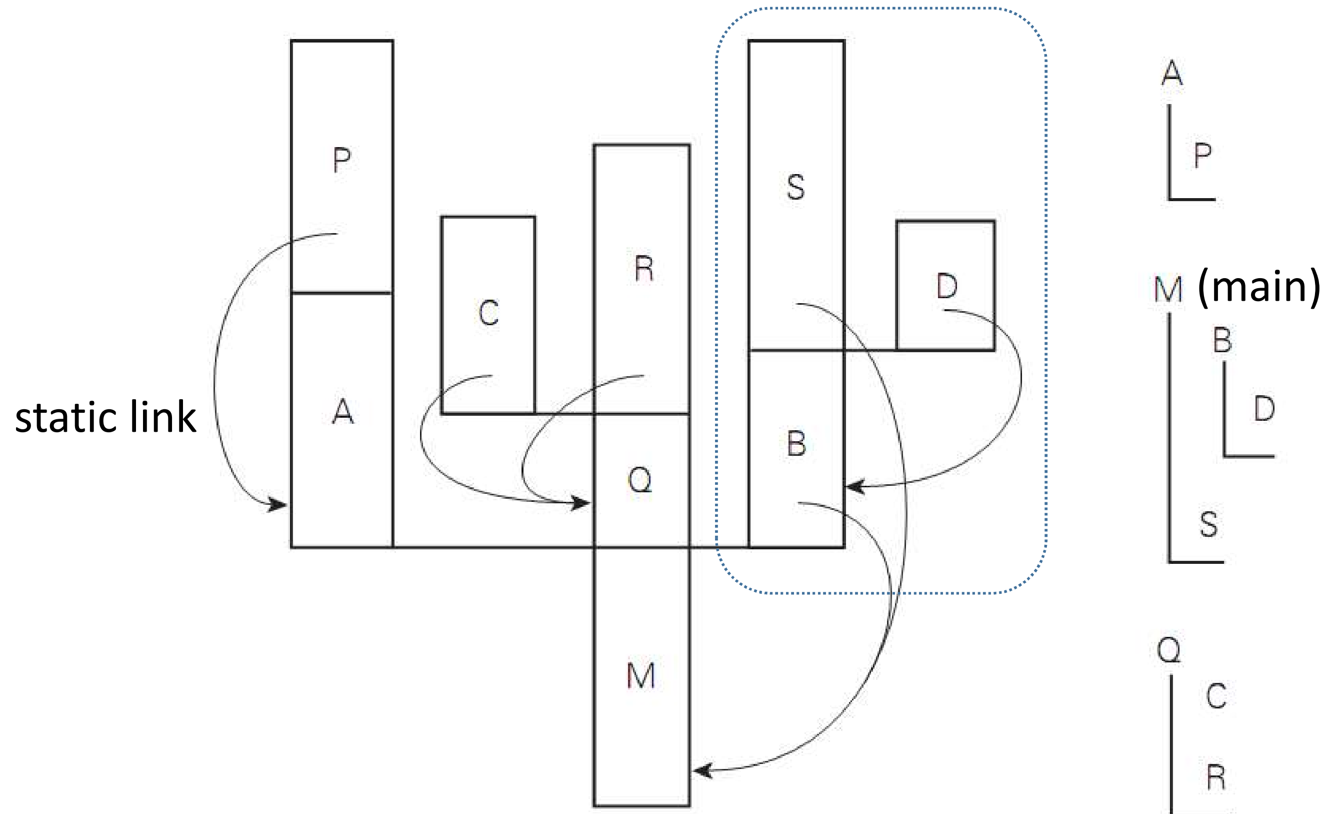
- Concurrently calling one another
  - Coroutines are execution contexts that exist concurrently, but that execute one at a time, and that *transfer* control to each other explicitly, by name

- Coroutines can be used to implement
  - Iterators
  - Threads

- Separate stack should be maintained
  - Because they are concurrent (i.e., simultaneously started but not completed), coroutines cannot share a single stack

Multiple Stacks for Coroutines

B calls subroutine S and created coroutine D



Cactus Stack

Summary

- Function
  - Stack frame
  - Passing parameters
- Exception
  - Unexpected/unusual HW/SW condition caused by current program
  - Try-catch (Java, C++)
- Event
  - HW/SW condition required to respond by current program
  - Event handling mechanism
- Coroutine
  - Concurrent execution of multiple sequences, but execute one at a time
  - Transfer:
    - *non-local goto* to jump to other coroutine's last transfer (continuation)
  - Cactus stack: Multiple concurrent stacks