Notes for CSC 254, 25 and 27 Mar. 2025

==========

Stack management

Recall allocation strategies: static, stack, heap

Maintaining the Run-Time stack

- Contents of a stack frame
  - bookkeeping: return PC (dynamic link), saved registers, static link,
    - (rarely) alignment or interrupt mask information
  - arguments and return value(s)
  - local variables
  - temporaries
- Maintenance of stack is responsibility of "calling sequence"
  and subroutine "prologue" and "epilogue".
  - space is saved by doing as much work as possible in the prologue and epilogue
  - time ***may*** be saved by doing work in the caller instead, where more
    - information may be known. E.g., there may be fewer registers
    - ***in use*** at the point of call than are used ***somewhere*** in the callee.
  - common strategy is to divide registers into "caller-saves" and
    - "callee-saves" sets.
    - Caller uses the "callee-saves" registers first;
      - "caller-saves" registers if necessary.
    - Callee uses the "caller-saves" registers first;
      - "callee-saves" registers if necessary.

- Local variables, parameters, and temporaries are assigned fixed ***offsets*** from the frame pointer or stack pointer at compile time

- Variable-length locals and parameters are handled with descriptors (dope vectors—covered in Chapter 8). The descriptors are at known offsets. For locals, they are accompanied by a pointer to space higher up in the frame. For value arguments, the pointer points down in the frame.

Stack layout varies significantly from machine to machine and, to some degree, from compiler to compiler. Many compilers, for example, access everything relative to the

stack pointer when they can, so the frame pointer can be used for something else. This is not possible w/ variable-sized data in the frame.

Typical modern compiler aims to minimize memory accesses and to rely on simple instructions:

- no special instructions other than `jsr` or `call`
- most arguments passed through registers (but space reserved on stack)
- often skip frame pointer
- relatively stable `sp` (arg build area)
- simple leaf routines make no use of memory at all

Older compilers often used the stack more, and leveraged complex, special-purpose instructions:

- special subroutine-calling instructions to save and update the frame pointer, save registers, branch, and allocate space for the frame, all in one or two instructions
- special `push` and `pop` operations to load/store and update `sp` in one instruction
- (usually) all arguments passed on the stack
- (usually) real frame pointer
- (usually) `sp` moves up and down as arguments are pushed and popped.
  Convenient for function calls embedded in argument lists.
  No longer done this way on x86, however—x86-64, esp., makes more use of (now more numerous) registers.

Case study

The text presents LLVM on ARM-32 (e.g., iPhone) and GCC on x86 (32 & 64). I'll focus here on GCC on x86-64.

register usage

16 64-bit integer registers, 16 128-bit FP/SSE registers
(various other legacy registers that are not commonly used)

naming of registers is complicated, due to evolution of the ISA over the years

`rsp`	stack pointer; callee-saved
`rbp`	frame pointer (if used); callee-saved
`rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9` (in that order)	first 6 integer arguments; caller-saved

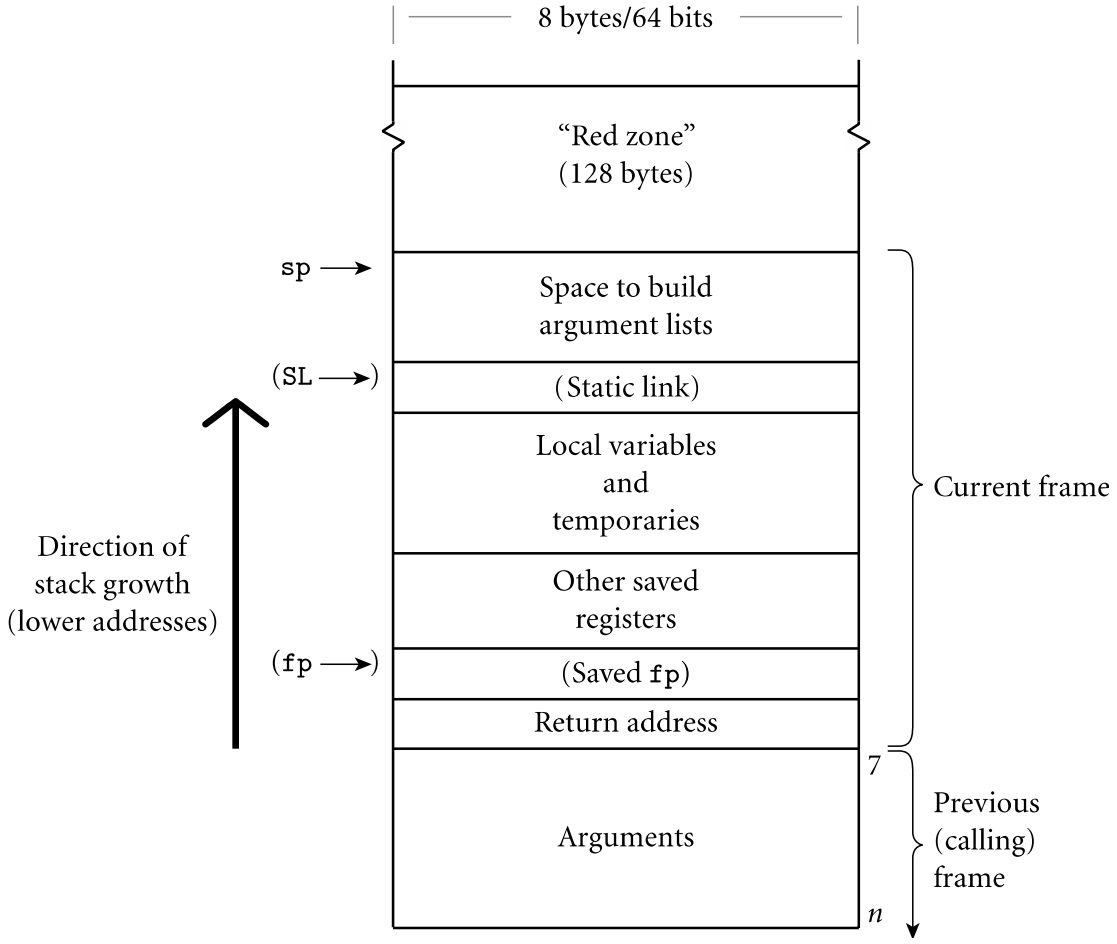
`rbx, r12, r13, r14, r15`	callee-saves temporaries
rax, r10, r11	caller-saves temporaries

static link (if needed) is passed in `r10`.

`rax` and (if needed) `rdx` are used to return function value

`rax` and `rdx` are over-written by division operation

several other similar special cases—non-orthogonal architecture



Note: In previous incarnations of x86 ABI, SP points to last used location.

On some machines/OSes, it points to first ***unused*** location. ***Beware!***

Actual calling sequence

Caller

1) saves caller-saves registers into temporary locations in current frame, if necessary
2) puts args into registers and (if necessary) the build area at the top of the current frame
3) puts static link in `r10` (skipped for C, or for level-0 callees)
4) executes call

In prologue, Callee

1) pushes `fp` (decrementing `sp` by 8)
2) copies `sp` into `fp`, creating new `fp`
3) pushes callee-saves regs, if necessary
4) subtracts rest of frame size from `sp`

In epilogue, Callee

1) sets return value, if any
2) restores callee-saved regs, if any
3) copies `fp` into `sp`, deallocating frame
4) pops `fp` off stack
5) returns

Steps 3) and 4) can be combined into a one-byte '`leave`' instruction. It's never been entirely clear to me why compilers sometimes generate it and sometimes don't—perhaps details of timing on particular processor implementations.

After call, Caller

1) moves return value from register to wherever it's needed (if appropriate)
2) restores caller-saves registers lazily over time, as their values are needed

Many parts of sequence can be elided in special cases.
In particular

- many routines get by w/out `fp`
- red zone lets small leaf routines avoid updating `sp`

Access to non-local variables via static links

Each frame points to the frame of the (correct instance of) the routine inside which it was declared. In the absence of formal subroutines, "correct" means closest to the top of the stack.

You access a variable in a scope *k* levels out by following *k* static links and then using the known offset within the frame thus found.

You set up static links as follows:

case 1: callee is nested (directly) inside you

callee's static link is pointer to your frame

case 2: callee is *k* scopes out (*k* may be 0)

callee's static link is found by indirecting off your own static link *k* times

Procedures as parameters: When you form the closure, you figure out a static link just as if you were going to call the routine directly; the closure consists of the routine's address and the static link.

==========

Parameter passing

Three basic implementations:

value, value/result (copying—when you have a value model of variables)

reference (aliasing—maybe even if you normally have a value model)

closure

Closures used not only for formal subroutines, but also ***name*** (lazy) parameters and (ancient) label parameters (Algol 60, 68)

Some languages (e.g., Pascal) have provided `val` and `ref` directly.
Problem: pass big thing by `val` or `ref`? —speed v. safety tradeoff
Solution? (Modula-2): '`const`' mode that is read-only but passed by reference

but then `val` and `const` for small things are ~semantically redundant

Ada went for semantics: who can do what

`in`	formal initialized; actual not modified
`out`	formal not initialized; actual modified
`in out`	formal initialized; actual modified

Ada `in out` is always implemented as value/result for scalars, and either value/result or reference for structured objects. The language manual says your program is "erroneous" if it can tell the difference.

Call by reference is the only option in Fortran. If you pass a constant, the compiler creates a temporary location to hold it. If you modify the temporary, who cares?

In a language with a reference model of variables (Lisp, ML, etc.), the obvious approach is to make the formal parameter refer to the same thing as the actual parameter. I like the name ***call by sharing*** for this, because it isn't clear whether to think of it as value (formal parameter is a copy of the actual) or reference (formal parameter is a reference, and can change from caller's perspective). Note that with call by sharing you can change the value of the referred-to thing (assuming it isn't immutable), but you can't change which thing is referred to.

Call-by-name is an old Algol technique. Think of it as call by textual substitution (a procedure with all name parameters works like a macro). What you pass are hidden procedures called ***thunks***.

Jensen's device example:

```
function sum (expr, index : name real; low, high : const integer)
   returns answer : real;
begin
   answer := 0;
   for i in low..high loop
      index := i;
      answer +:= expr;
   end loop;
end sum;

S := sum (A[2*i-1], i, 1, 10);
```

(Curiously, it doesn't seem to be possible to write a general-purpose swap routine with name parameters.)

Call-by-name is a naive implementation of normal-order evaluation. ***Call-by-need*** does memoization. It's used in Haskell, which is purely functional, and in R, which is not. Both call-by-name and call-by-need are considered "lazy evaluation" by the functional programming community (there's some inconsistency of nomenclature here—compiler people sometimes use "lazy" only for call-by-need).

In a pure functional language call by name and call by need are semantically indistinguishable; with side effects they aren't.

Note that passing dynamic arrays by value is tricky. The actual parameter list on the stack contains a fixed-size dope vector and a pointer, but where does the data go? Most likely option is below the arguments.

Summary:

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†

* 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.

Other parameter issues

- conformant arrays—variable-size array parameters in a language that otherwise doesn't support variable-size arrays
- default (optional) parameters—don't avoid cost
- named parameters—great for long parameter lists
- variable number of parameters—typesafe?

Function returns

Pascal and Fortran return values from functions by assigning to the function identifier. User cannot re-use the name of a function inside. Later languages tend to have an explicit 'return' statement (as in C or Ada), or a named return value (as in Algol 68, above, or Go).

Another advantage of named returns is that you can use the name inside expressions:

```
-- Ada
type int_array is array (integer range <>) of integer;
   -- array of integers with unspecified integer bounds
function A_max (A : int_array) return integer is
rtn : integer;
begin
   rtn := integer'first;
   for i in A'first .. A'last loop
      if A(i) > rtn then rtn := A(i); end if;
   end loop
   return rtn;
end A_max;


-- Go
func A_max(A []int) (rtn int) {
   rtn = A[0]
   for i := 1; i < len(A); i++ {
      if A[i] > rtn { rtn = A[i] }
   }
   return
}
```

==========

Exceptions and Events

What ***is*** an exception?

- an unusual condition detected at run time
  Examples:
  - arithmetic overflow
  - end-of-file on input
  - wrong type for input data
  - user-defined conditions (not necessarily errors)
- error v. nonlocal return—different mechanisms? (Common Lisp)

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
- allow user to handle errors without having to check these conditions explicitly in the program everywhere they might occur

Downsides

- may discourage careful checking of boundary conditions (laziness)
- introduces brittleness: caller has to be prepared to handle error
- Some companies (e.g., Google) have banned the use of exceptions in their code; Rust chose not to provide them

Consider handling of errors in a (large) recursive-descent compiler. It's something of pain in languages without exceptions: need extra parameters and checks in every procedure. May be simpler to be able to back out to exactly where you want to.

Pioneers:

PL/I: dynamically scoped
CLU: statically scoped, procedure/abstraction oriented
(can't handle locally)

Convergence in modern languages on built-in, statically scoped, "replacement" model: Ada, C++/Java/C#, ML, Common Lisp, Swift/Kotlin/Scala, Python/PHP/Ruby
Discussion here is for C++/Java.

Handlers local to code in which exception is raised

```
try {
   ...
      // throw obj;
   ...
} catch (end_of_file) {
   ...
} catch (io_error e) {
   ...
} catch (...) {
   ...                  // catch-all
}
```

Handlers must be at the end of a block of code (but can put blocks around any statement). ML and Common Lisp allow handlers on arbitrary ***expressions***.

Notion of ***matching*** an exception. Arguments as members of object.
(Ada has no arguments; ML makes them look like variant fields.)

Static (nested) binding within subroutine, then propagate up dynamic chain.
All functions that the exception propagated out of are terminated.
C++ executes destructors as appropriate on the way out.
Execution continues after handler code (which is always at the end of a block)

Implementation of statically-scoped exceptions

(1) Push handler address when entering a protected block, pop when leave. Sometimes implemented this way in C++, probably to minimize size and complexity of the run-time library.
(2) Do everything via lookup in tables produced by the compiler. This is the "right" way to do it.
(3) Can sort of fake it in C with `setjmp()` and `longjmp()`. These take state snapshot and restore on throw. Doesn't work right for non-volatile local variables. Very expensive.

Events

Commonly used in interactive programs. In Java:

```
class PauseListener implements ActionListener {
   public void actionPerformed(ActionEvent e) {
      // do whatever needs doing
   }
}
...
JButton pauseButton = new JButton("pause");
pauseButton.addActionListener(new PauseListener());
```

Or, with anonymous inner classes:

```
pauseButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      // do whatever needs doing
   }
});
```

This is a little ugly because Java insists on making everything a class. What you really want is a lambda expression to be executed when the event occurs.

In C# you have ***object closures***, which function fort of like lambdas. They're actually more like Runnables in Java or objects with an `operator()` method in C++. They are supported by a syntactic mechanism that C# calls ***delegates***—sort of syntactic sugar for single-method interfaces:

```
void Paused(object sender, EventArgs a) {
   // do whatever needs doing when the pause button is pushed
}
...
Button pauseButton = new Button("pause");
pauseButton.Clicked += new EventHandler(Paused);
```

or

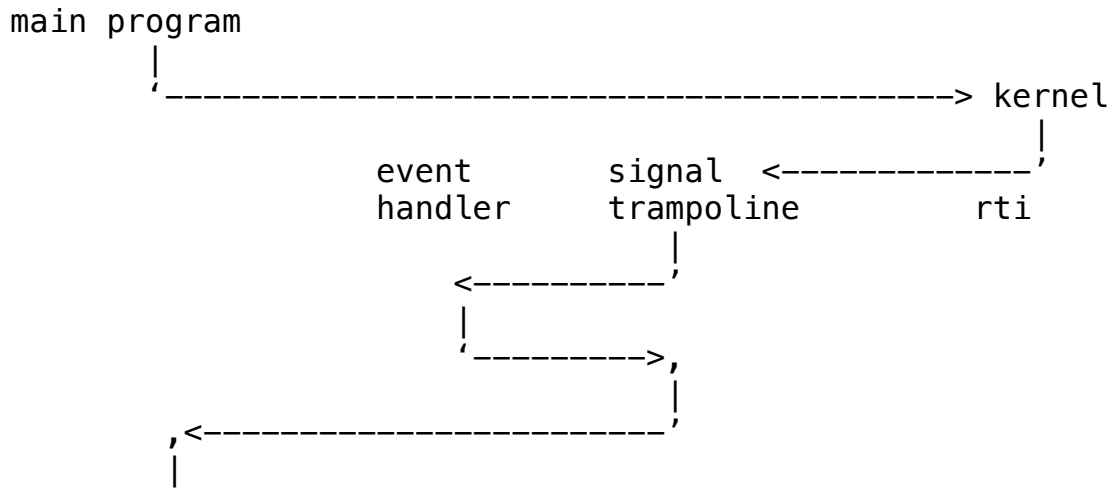
```
pauseButton.Clicked += delegate(object sender, EventArgs a) {
   // do whatever needs doing
}
```

Paused matches the pattern established by a delegate declaration in the library. Can be a static method or a method of a particular object—even a nested object. In the latter case, C# (like C++) allows access to static members of the surrounding class only. Java allows access to nonstatic members, but doesn't have delegate sugar.

Signal handlers

In Swing, as in many GUI packages, events are handled by separate threads, managed by the runtime. If they access data for which consistency is an issue, you have to use explicit synchronization.

In other languages, event handling can happen in the currently running thread, as if it were a spontaneous subroutine call. Because the caller doesn't actually make the call, we need extra machinery, called a ***trampoline***, to fake the calling the sequence:



Note that because the main program and signal handler run on the same thread, they can't use scheduler-based locks to synchronize access to shared data structures. Rather (as we saw in the implementation of thread scheduling in Chap. 13), the main program needs to turn off signals while manipulating data that may also be accessed in a handler.

Note also that the word "signal" is used in Chap. 13 for a second, *unrelated* purpose—the `signal` and `wait` operations of Java monitors. Don't let that confuse you.

Asynchronous programming

Allow a single thread to maintain, and switch among, multiple execution contexts.
For the purpose of accommodating waits for asynchronous (external) events like I/O, subprogram completion, or timers. Commonly used to structure multi-step activities:

- get a request
- look something up in a database
- interact with a credit card or shipping agency
- ...
- send response

The standard event-handling mechanism of JavaScript/TypeScript.
Also supported in C#, F#, C++, Haskell, Rust, Kotlin, Swift, Python, ...

Consider JavaScript as an example. Early (still supported) syntax:

```
function callbacks(e) {
    let n = 0;
    setTimeout(a => {
        e.putInt(++n);
        setTimeout((b) => {
            e.putInt(b);
            setTimeout((c) => {
                e.putInt(c)
            }, 2000, b+1)
        }, 2000, a+1)
    }, 2000, n+1)
}
```

The `setTimeout` routine takes a lambda as argument, which will be executed after the timeout (2nd arg) expires. The third arg is the parameter to be sent to the callback. Note that `setTimeout` returns immediately; the callback is scheduled for the future. A similar callback might be triggered not by a timer but by the response to a query sent to some remote server. (Note that we didn't really need a, b, and c in the code above, since n is visible. If the callbacks were separate outermost functions, n wouldn't have been visible to them all. In subsequent examples I'll omit this third [optional] arg.)

So how does this execute? There's a single thread of control in the JavaScript runtime. It keeps executing until it runs off the end of its code. Then it returns into the runtime

system. If any callbacks have been created, the runtime waits until one or more of them are triggered, and uses the single thread to execute them, possibly scheduling additional callbacks. At the end of every handler, the thread returns to the runtime and the cycle continues.

Callback-based code tends to be a mess of spaghetti. To clarify the connections among related functions, ECMAScript 2015 introduced Promises:

```
function twoSecDelay() {
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(), 2000)});
}

function promises(e) {
    let n = 0;
    twoSecDelay().then(() => {
        e.putInt(++n);  return twoSecDelay ();
    }).then(() => {
        e.putInt(++n);  return twoSecDelay ();
    }).then(() => {
        e.putInt(++n)
    })
}
```

A Promise is an object that represents the value to be generated by an asynchronous action (e.g., the expected response from a server). Here the `twoSecDelay` function returns a `Promise` object whose constructor set a timeout. The `then` method attaches a handler to a list in the Promise. The timeout handler calls the (built-in) resolve function, which calls everything in the list. In this example, function `promises` executes quickly, without waiting for anything.

The programmer writes the executor function that is passed to the Promise constructor. The constructor (internal to the library) calls the executor, passing as parameters a pair of (again, internal) functions that the executor should arrange to call when the asynchronous action completes: one (`resolve`) for when it completes successfully (passing the action's value as parameter); one (`reject`) for when it completes unsuccessfully (passing an error message as parameter). In our code above, timeout doesn't have a failure case: the Promise constructor simply sets a timeout and tells it to call resolve when the time has expired. A second argument to `then` (not used here) attaches handlers to a second, failure-case list in the `Promise`.

ECMAScript 2017 introduced `await`. This builds on Promises, making them even easier:

```
async function awaitAsync(e) {
    let n = 0;
    await twoSecDelay();
    e.putInt(++n);
    await twoSecDelay();
    e.putInt(++n);
    await twoSecDelay();
    e.putInt(++n);
}
```

Note that `twoSecDelay` returns a Promise, just as it did before. But instead of attaching handlers to it with `then`, we simply `await` it. The *continuation* of the `async`, after the `await`, takes the place of the lambda that would be passed to `then`. The implementation is similar to that of true iterators; it leverages the fact that `await` can occur only in an `async` routine, just as `yield` can occur only in an iterator. This means that any stack space used by an `async` routine is freed before the `async` waits, so a single stack continues suffice for a single-threaded program with an arbitrary number of `async`s.