CS 354 Spring 2021 (Park)

RUN-TIME STACK OF A PROCESS AND FUNCTION CALL CONVENTION

1. Function calls vs. goto spaghetti code

In many high-level programming languages, including C, writing code using functions (or procedures) is encouraged. The opposite of this modular approach would be using "goto" statements (i.e., branches) to write software, also referred to as spaghetti code, which Dijkstra championed against in the 1970s.

Since the instruction set of CPUs don't contain "function calls" (e.g., the "call" instruction in x86 is equivalent to two instructions -- save the program counter followed by a jump/goto) the elegance and abstraction afforded by high-level programming languages (say C) that support function calls with argument passing and return value must be mapped to "goto" statements when a C program composed of multiple functions is translated into machine code by a compiler (e.g., gcc).

Most of this part is hidden from the programmer since it is automatically handled (i.e., extra bookkeeping code for coordination is injected) by a compiler.

2. Caller-callee context switching

When transferring from caller to callee, it is a form of "context switching" in the sense that a process's working environment has switched from one part of the code (caller function) to another (callee function). In operating systems, we are interested in a more advanced form of context switching where we switch from one process to another. That is, it is not that we are going from one function in a process to another function in the same process, but we are allocating the CPU from one process to another. This key responsibility of a kernel (a more technical name for OS which sometimes is used loosely), called scheduling, utilizes similar mechanisms to function call context switching hence knowing basic function call mechanics is essential.

3. What is a run-time stack

When a program is readied to run on a CPU and becomes a process, it is important to make the process run efficiently. An overhead that is incurred by the use of function calls is that caller and callee (foo() and hoo() in the example described in class) must agree on a convention to pass arguments from caller to callee. Same goes for returning a value computed by the callee to the caller, saving the registers used by foo() before branching to hoo() so that they may be restored upon completion of hoo(). Note that many registers are shared.

To reduce the overhead incurred by coordination of caller and callee, hardware and software support is provided to facilitate correct and efficient coordination through a data structure called a run-time stack.

Each process has its own private run-time stack.

The specifics of what a run-time stack looks like depend on hardware, OS, and compiler. For example, Cisco's Linksys E2100L's MIPS CPU (Atheros 9130, now part of Qualcomm) has a dedicated stack pointer (SP) register to hold the address of the top of the run-time stack of a process, but it does not have a frame pointer (FP) register that holds the beginning address of the run-time stack of a process. Our Intel x86 CPUs support both, called ESP ("E" indicates 32-bit x86 architecture) and EBP (BP stands for base pointer which is Intel jargon for frame pointer).

4. Dependencies on x86 backends in the XINU lab

In the XINU lab, we use Galileo x86 backends which are single-core (i.e., single CPU) 32-bit machines. In the case of multi-core 64-bit x86 machines (e.g., our frontend Linux PCs)

64-bit refers to the size of buses and registers. Registers in the 64-bit x86 PCs are prefixed by a "R" (e.g., RSP instead of ESP). 64-bit machines can be operated in 32-bit mode (a common mode for many apps) which is supported by Intel hardware through backward compatibility. For example, when using ESP in place of RSP in a 64-bit machine, ESP refers to the lower 32 bits of the 64-bit RSP register.

In x86 machines, with the help of dedicated SP and BP registers, and additional hardware instructions for accessing a stack (push/pop) and and returning from callee to caller (ret instruction), code is generated by compilers to coordinate function call bookkeeping between caller and callee. Thus almost everything is done in software (compiler) with some support from hardware (dedicated registers SP/BP, stack instructions push/pop), and ret to restore program counter (PC). Other instructions including call and leave are equivalent to a sequence of instructions aimed at providing more convenient assembly language programming.

5. Logical structure of a program

In the C language convention, we said that a program is laid out in memory (view it as RAM although we will see later that memory of processes is virtualized in many systems) as

```
text data heap --> . . . <-- stack
```

where text (or code) is located in low memory (say location 0) and stack is located in high memory (say maximum RAM address, e.g., 4G) and grows downward (when viewed vertically). The data area is composed of initialized and uninitialized parts whose exact shape and boundaries we will inspect in the first lab assignment.

6. Caller-callee coordination

When a program runs, the run-time stack maintains bookkeeping information for caller-callee coordination. The bookkeeping information for the calling function is called caller stack frame, and analogously callee stack frame for the called function. As discussed in class, the boundary of each stack frame is delineated by SP and BP. BP contains the starting address of the stack frame of the callee function and SP points to the top of its stack frame (and, by implication, the top of the shared stack).

Since there is only pair of SP and BP registers, the caller's BP value will need to be saved when the callee is invoked. The caller's SP value will not need to be saved since it can be inferred from the callee's BP value (the start of the callee's stack frame is the end/top of the caller's stack frame). The generic format, also used in x86 with C compilers -- this format or convention is called C declaration (CDECL) -- was discussed in class. For the following simplified foo()/hoo() example,

```
int foo(void) {
int         x;
            x = hoo(5,10);
}

int hoo(int a, int b) {
int r;
            r = a + b;
            return(r);
}
```

the run-time stack (growing from high-to-low address, depicted right-to-left below) looks like

```
            … | r | caller's saved BP | caller's saved IP | 5 | 10 | …
```

where SP points to the address of the local variable r of hoo() which is at the top of the run-time stack, and BP points to the caller's saved BP location.

Part of the assembly code generated by a C compiler will look like

```
foo:                                  # part of foo
```

```
push       10
push       5
call       hoo                    # call pushes the caller's IP then jumps to label hoo

hoo:                              # part of hoo

push       BP                     # saves caller's BP
mov        SP, BP                 # copies SP to BP which sets hoo's base to foo's top
mov        8(BP), AX              # moves hoo's 1st argument 5 to register AX
                                  # we assume integers are 4 bytes long
                                  # note that BP serves as a useful anchor point
mov        12(BP), BX             # moves the 2nd argument 10 to register BX
add        AX, BX                 # adds values in AX, BX and stores the result in AX
mov        AX, -4(BP)             # copies the added value 15 to local variable r
leave                             # leave is equivalent to mov BP, SP
                                  # which sets the SP to start of hop's stack frame and
                                  # pop BP which pops the saved BP into BP and
                                  # sets SP to point to the caller's saved IP
ret                               # loads IP with the value pointed to by SP
```

Note that after the ret instruction, the foo()'s stack frame still contains
the arguments 5, 10 passed to hoo(). Hence in this approach it's the
called foo()'s responsibility to clean up the arguments from its stack
frame. This implies that the saved IP points to clean-up code injected
by the compiler called epilogue code. In conjunction with bookkeeping
code injected by the compiler to switch from caller to callee (called
prologue code), the hardware and software dependent mechanisms allow
programmers to code in high-level languages such as C that support function
or procedure calls without worrying about how such high-level language
constructs are mapped to low-level machine code.

The above example resembles AT&T's assembly language syntax but has been
simplified to reduce unnecessary clutter. The alternative is Intel's
assembly language syntax which is not used in XINU.