

Using a Variable Number of Arguments

The `printf` function is a special type of procedure that takes a variable number of arguments. The procedure declaration requires a special syntax to defined the fact that beyond a certain argument, the number and type of the parameters cannot be checked at compile-time. Instead, the number and type of the parameters has to be computed at run time. In ANSI C/C++, you us the *ellipsis* in the type signature to denote a variable argument list:

```
void printf(const char* format, ... /* variable args */);
```

For example, suppose that you want an error message function that takes a format string and a variable number of arguments.

```
printf("hello, world");  
printf("length of '%s' = %d\n", str, str.length());  
printf("unable to open file descriptor %d\n", fd);
```

The `printf` procedure take a `const char*` as the first argument, and any number of arguments after that, where the type of the arguments and their number has to conform to the “format specification” that is encoded in the first argument using the special “%” escaped characters. For example:

- `%s` denotes a string argument (`const char*`). `%ws` denotes a wide (`wchar_t`) character string
- `%d,i,o,u,x,X` all denote an integer argument
- `%f` denotes a double argument
- `%c` denotes a character argument, `%wc` denotes a “wide” character (`wchar_t`)
- `%%` denotes a `%`, no argument is expected
- `%p` denotes a pointer value (`void*`)

Integer and floating point formatting can also include an integer precision specification. See the man page for `printf` for the complete list of format specifications.

Implementation of Variable Argument Printf Procedure

The following special “functions” are defined in `<stdarg.h>` and are used to process the variable arguments. The `printf` function effectively computes its arguments at run-time, and applies the appropriate type conversion of the argument based on the format specification.

```
void va_start(va_list, lastarg);
type va_arg(va_list, type_of_arg);
void va_end(va_list);

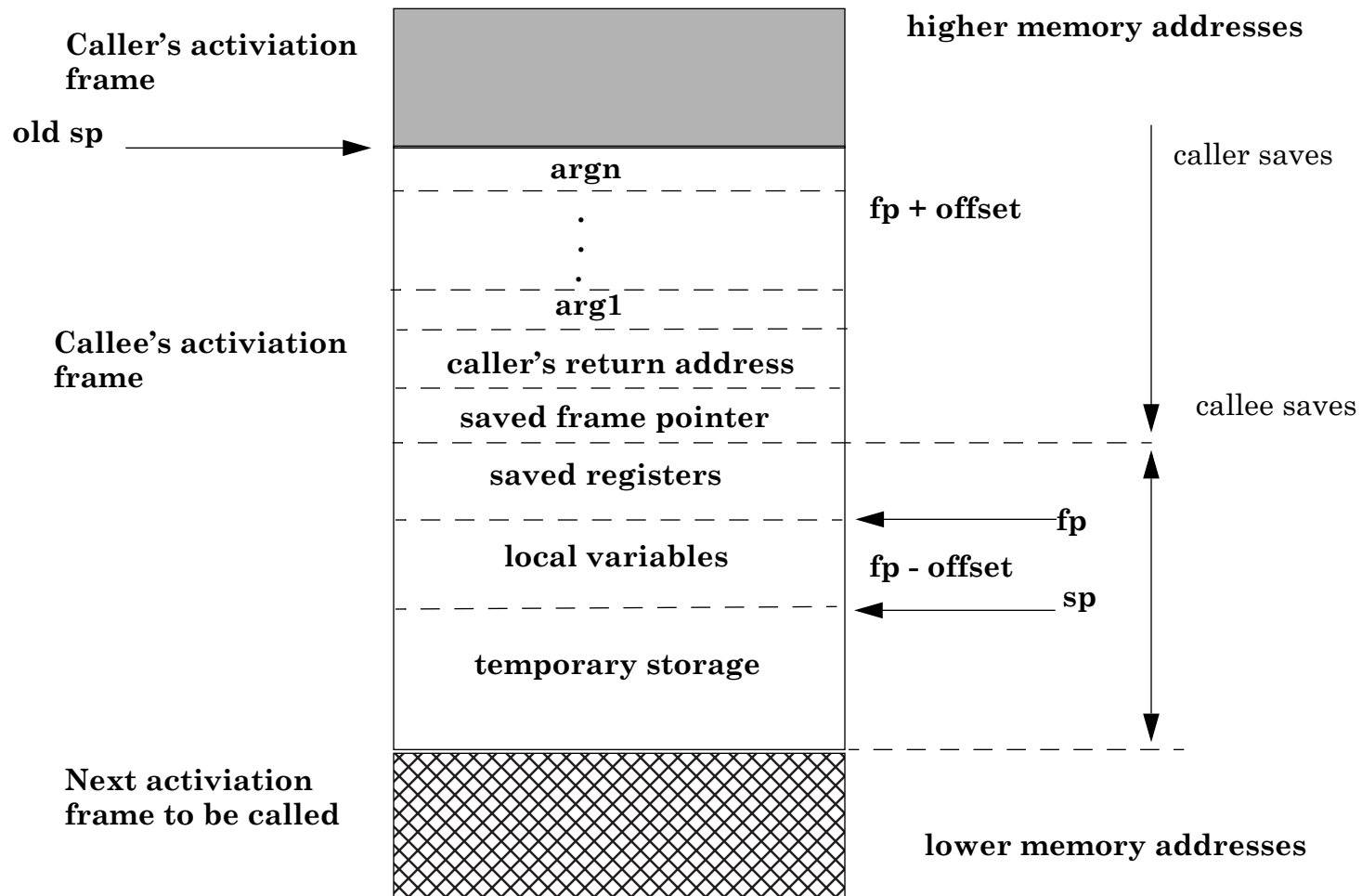
void printf(const char* format, ...)
{
    int i; char c; char* s; double d;
    va_list ap; /* declare an "argument pointer" to a variable arg list */
    va_start(ap, format); /* initialize arg pointer using last known arg */

    for (char* p = format; *p != '\\0'; p++) {
        if (*p == '%') {
            switch (*++p) {
                case 'd':
                    i = va_arg(ap, int); break;
                case 's':
                    s = va_arg(ap, char*); break;
                case 'c':
                    c = va_arg(ap, char); break;
            }
            ... /* etc. for each % specification */
        }
    }
    ...

    va_end(ap); /* restore any special stack manipulations */
}
```

Stack Activation Frame for Var Args Procedure

The caller “knows” the number of arguments pushed onto the stack, but the called procedure (e.g., printf) does not, so it has to compute them at run-time. The `va_start` procedure computes the `fp+offset` value following the argument past the last known argument (e.g., `const char* format`). The rest of the arguments are then computed by calling `va_arg`, where the ‘`ap`’ argument to `va_arg` is some `fp+offset` value.



Concurrent Programming using Threads

Threads are a language control mechanism that enable you to write concurrent programs. You can think of a thread in an object-oriented language as a special kind of “system object” that contains information about the state of execution of a sequence of function calls that are said to “execute as a thread”. Usually, a special “run” or “start” procedure starts a separate thread of control.

Normally, when you call a function or procedure, the compiler sets-up a stack activation frame on the run-time procedure call stack, pushes arguments (or puts them into registers), and ‘calls’ the function by branching to the address of the function. As we have seen previously in the course, the activation frame for a procedure is used as temporary storage for locally allocated variables declared in the scope of the called procedure.

In a **sequential program**, there is only one run-time stack and all activation frames are allocated in a nested fashion on the same run-time stack, corresponding to each nested procedure call. In a **concurrent program** (also called a **multi-threaded** program), each “thread” represents a separate run-time stack, so you can have multiple procedure call chains running at the same time, possibly on multiple processors.

For example, C/C++ with POSIX threads and Java on Solaris supports multi-processor threads and NT Threads on Windows NT supports multi-processor threads.

In a sequential program, the main run-time stack is allocated at program start and all procedure calls, including the initial call to “main” are made on this single run-time stack. In a concurrent program consisting of separate threads of control, a program starts on the system run-time stack where the main procedure runs. Any functions/procedures called by the main procedure have their activation frames allocated on this run-time stack.

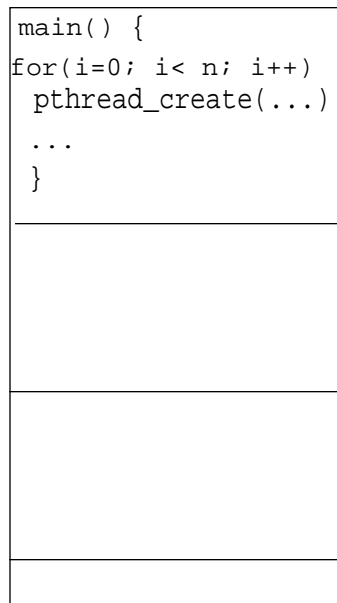
If the main procedure creates a new thread to run some procedure concurrently with the main thread, then a new run-time stack is dynamically allocated from the heap and the activation frames for the procedures are allocated on this new stack.

Thread Stacks

Question: How large should a heap allocated thread stack be?

A thread stack will contain the activation record of the “starting” thread procedure (e.g., called the “run” method in Java), as well as any procedures that are called by the procedure that was first started in the new thread of control. So, the thread stack needs to be large enough to hold the maximum number of bytes required to hold all the activation records of the deepest procedure call chain, as well as storage for all local variables allocated on the stack, and usually some book-keeping information about the thread itself.

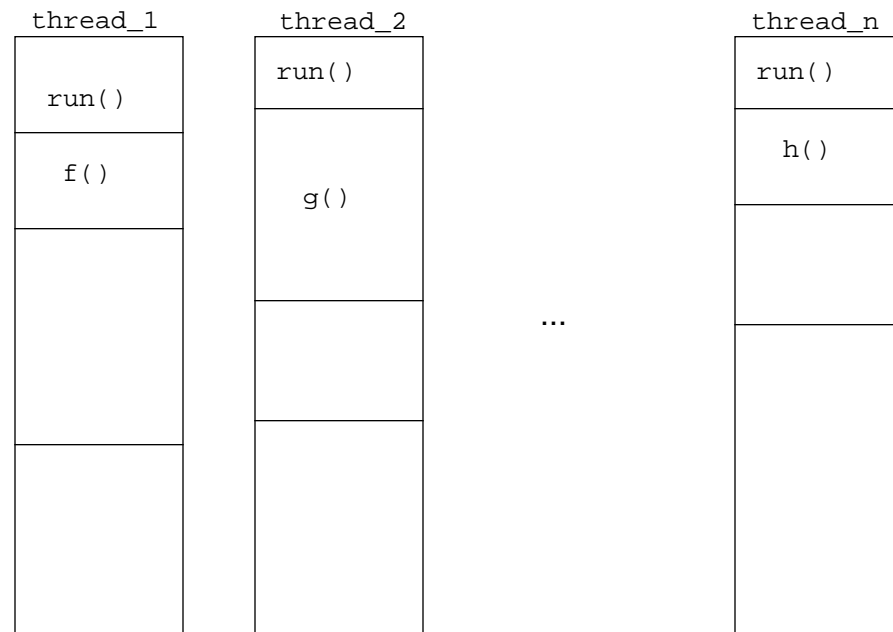
Main thread and run-time stack



Stack
Activation
Frames



Multiple thread run-time stacks, each a separate “thread of execution”



Thread Stack Size

On operating systems that support processes with multiple threads of control, threads stacks are typically set at 1MB, consisting of contiguous virtual memory pages, that are allocated incrementally at run-time by the system. There may also be some extra pages at the “top” or “bottom” used to store some thread book-keeping information. There is also usually an extra virtual memory page allocated “above” the top and “below” the bottom of the stack to detect underflow/overflow conditions. The overflow page is called the *red-lined* region of the thread stack. A red-lined page is achieved by asking the virtual memory system of the operating system to mark the page(s) for the red-lined region as non-writable, non-readable, and non-executable, so that any attempt to access memory addresses in the red-lined region will cause a memory protection trap generated by the memory management unit of the machine, causing the operating system to terminate the running program (with a Segmentation Violation on Unix and a General Protection Fault “blue screen” on Windows).

For example, if a recursive procedure call does not terminate, it would eventually cause a new stack activation frame to be allocated on the thread stack that caused the stack pointer (SP) to refer to a memory address in this redlined stack region. Any attempt to read or write an address in this region would result in a memory access violation.

