CI583: Data Structures and Operating Systems
Basic OS Concepts

Outline

Terminology

We need a few basic terms before we can begin, many of which you may already be familiar with:

- CPU: the central processing unit is the piece of hardware that executes instructions one at a time to carry out basic arithmetic, logic and input/output (IO) operations.
- Register: a small amount of storage inside the CPU which can hold one word (normally 32 or 64 bits). A modern CPU has access to 32, 64 or more. The fastest type of storage. The CPU moves data in and out of registers in order to carry out arithmetic operations etc.

Terminology

- Assembly code: written in low-level but still human-readable programming languages whose instructions map very closely to actual machine instructions. More or less all you can do in assembly is move values in and out of registers and the stack and call arithmetic operations. Each assembly language is specialised for particular hardware. High level languages are often compiled into this.
- Process: an instance of a program that is currently being executed. Each process may continue several threads, each of which may run concurrently.

Terminology

- Primary storage: also called main memory or RAM. A form of volatile data storage used to hold the processes currently being executed, parts of the operating system itself, and so on.
- Secondary storage: also called external storage. Non-volatile data storage typically held on a hard drive. Far slower than primary storage, since accessing a particular location requires physical moving parts.

Terminology

- Stack: or call stack. An area of main memory dedicated to store information about a process, and which makes temporary storage available to that process.
- Heap: an area of main memory available for use by executing processes.

Context switching

As the OS transfers control between processes, interrupts processes to deal with an I/O device etc., it needs to keep track of the setting within which execution is taking place: the context.

The context includes code and data – everything required for (say) a process to continue its work.

Context switching

Work other than processes need contexts too.

For instance, a process, P, may spawn several threads, each of which share's P's code and some of its data, though each thread maintains local data too.

Context is switched between threads on the basis of time-slicing and the type of task a thread is carrying out – e.g. a thread may be blocked on I/O.

Context switching

Even within a single thread we need to think about switching contexts, though this is handled by the compiler. Consider the following code in a C-like language:

Compute powers

```
1   int main() {
2     int i;
3     int a;
4     //...
5     i = sub(a, 1);
6     //...
7     return(0);
8   }
9
10  int sub(int x, int y) {
11    int result = 1;
12    int i;
13    for(i=0; i<y; i++)
14      result *= x;
15    return(result);
16  }
```

The problem

Every time a function is invoked, the compiler needs to turn the invocation, eventually, into some assembly code that includes allocating new storage for the local parameters.

We push these local parameters onto the stack, the area in memory reserved for temporary values.

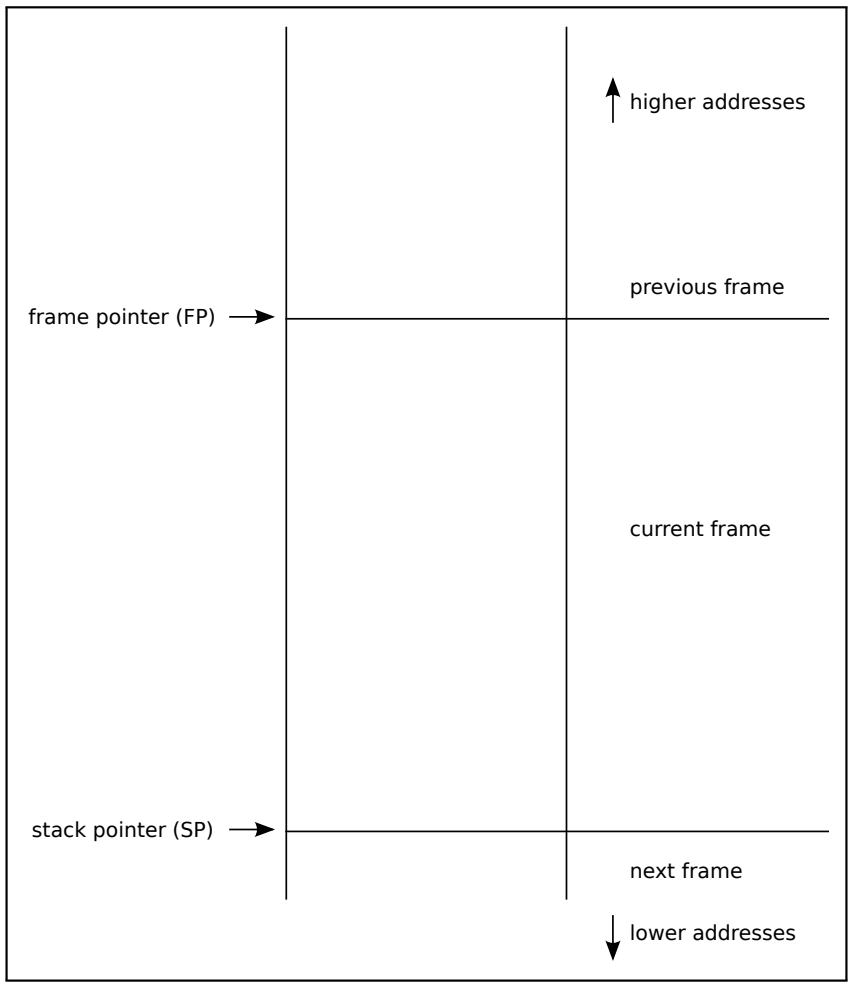
The answer: stack frames

Or, activation records

We keep local variables on the stack, but `pop` and `push` aren't convenient enough. We keep a pointer in a special register, pointing to the current head of the stack. This is the stack pointer. Then, the addresses of local variables can be given relative to the stack pointer.

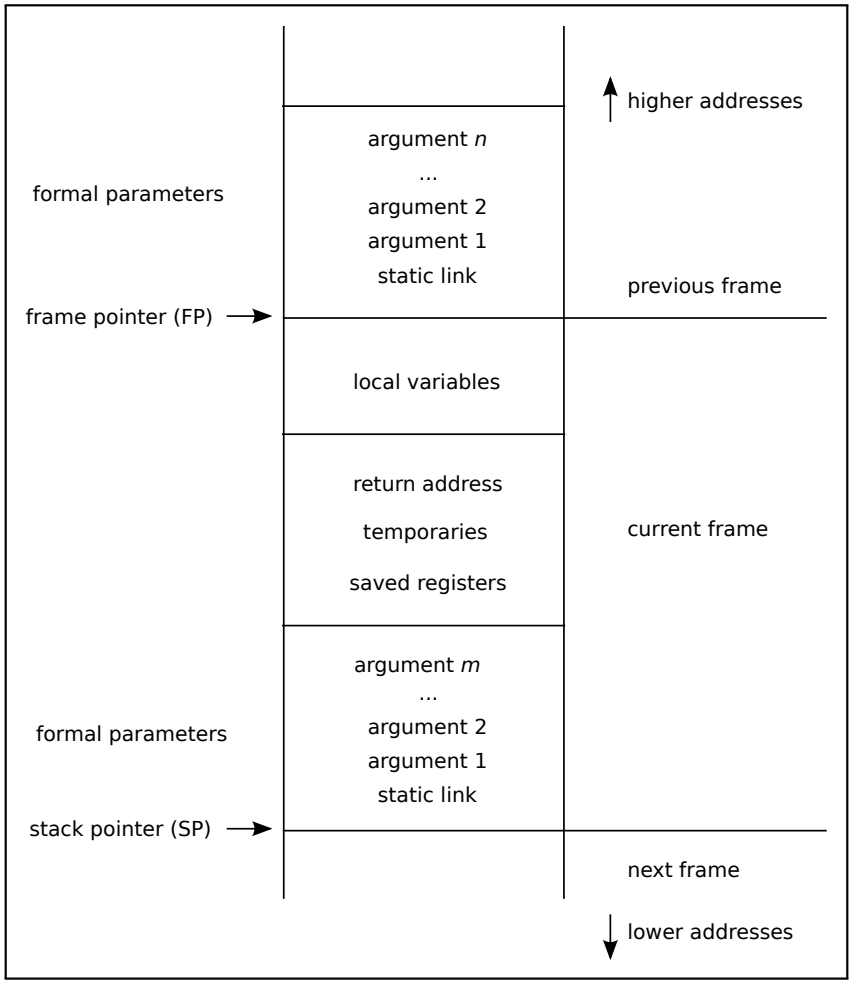
Everything beyond it is treated as rubbish. Rather than popping lots of elements when function invocations end, we can just change the value of the stack pointer.

Stack frames



Reproduced from Appel, *Modern Compiler Implementation in Java*, 2002.

Stack frames



Reproduced from Appel, *Modern Compiler Implementation in Java*, 2002.

Stack frames

There are lots of ways of organising a frame, but the standard way we have given allows us to interact "natively" with code generated by C compilers. (If we don't care about that, we could set up a different notion of frames.)

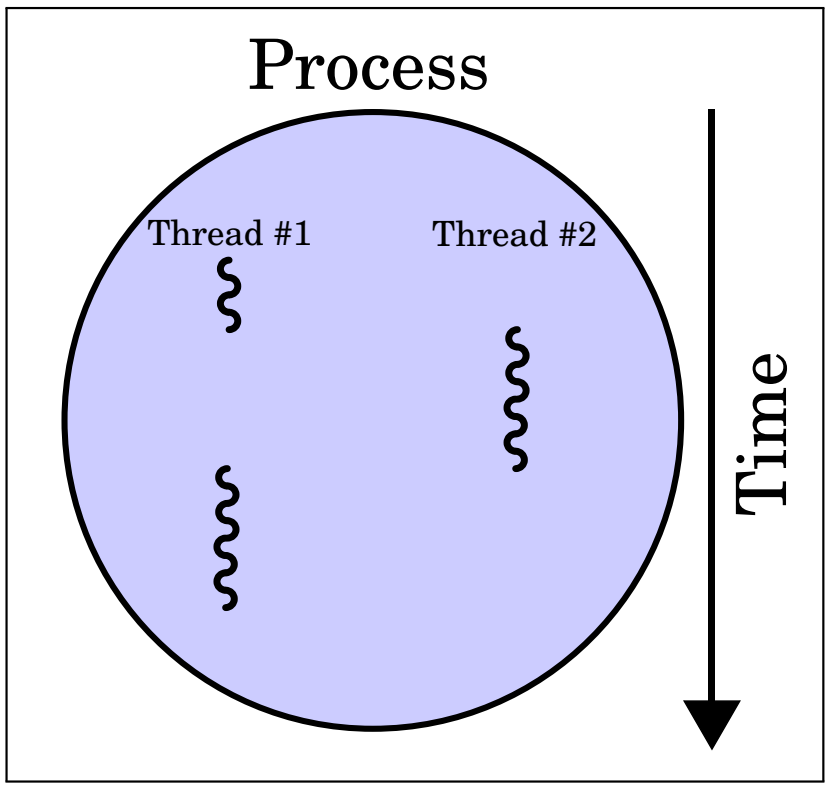
On entering a new activation record the stack pointer (SP) needs to be increased by the size of the current frame.

This means that the compiler has calculated how much storage is required by each method invocation. Then we can access local arguments relative to the value of SP.

Threads and coroutines

Conceptually, several threads executing within the same process are independent from one another, apart from the needs of synchronisation and cancellation.

However, to implement threads, we need one thread to directly pass control, or yield the processor, to another. We call such threads coroutines.



Threads and coroutines

The context of a thread is made up of its stack frame and the state of the registers (e.g. the current value of the stack pointer).

This information is stored in a data structure called a thread control block. To switch context between threads, we need to modify the active thread control block.

System calls

To protect the OS from attack, accidental or otherwise, almost all code runs in user mode, while the kernel, the heart of the OS, runs in privileged mode. Tasks which require privileged mode include using an I/O device and directly manipulating a memory location.

When our userland code wants to do one of these things, we ask the OS to do it for us using a system call. This is another example of a context switch, in this case switching between modes.

System calls

A couple of Linux system calls

%eax	**Name**	%ebx	%ecx	%edx	**Notes**
1	`exit`				Exits the program.
3	`read`	File descriptor.	Buffer start.	Buffer size (int).	Reads into the given buffer.
4	`write`	File descriptor.	Buffer start.	Buffer size (int).	Writes the buffer to the file descriptor.

The full set of these system calls make up the API of the OS.

System calls

To use these system calls we write the system call number to the register %eax then send an interrupt (pass control) to the OS:

Exiting a program in Linux x86 (Intel)

```
1 #...
2 movl $1, %eax #user mode, user stack
3 int $0x80 # switch to priv. mode and kernel stack
```

Higher level languages wrap this up in library calls, so that we just call `return`, say, but this is what is happening behind the scenes.

Interrupts

When an interrupt occurs, the processor puts aside the current context (that of a thread, or another interrupt) and switches to an interrupt context.

Interrupt contexts require their own stacks – not all are as simple as `exit`. The OS may need to keep track of state in order to handle the interrupt, which may execute asynchronously.

Interrupts

How to allocate a stack to an interrupt?

1. Allocate a new stack for each interrupt,
2. keep one interrupt stack to be shared by all, or
3. borrow the stack of the process or thread it is interrupting.

Interrupts

How to allocate a stack to an interrupt?

1. ~~Allocate a new stack for each interrupt,~~
2. keep one interrupt stack to be shared by all, **[VAX]**or
3. borrow the stack of the process or thread it is interrupting **[Most others]**.

Another approach, used by Solaris, is to categorise interrupts by the sort of state they need to maintain and to preallocate a stack for each level.