

## Compilers Course

### Lecture 17: Runtime Environment

Typical memory layout for a process:

```

|-----| high addresses
| Stack (dynamic, grows | compiler generates code to manage this
| down)                 |
| ...                   |
| Heap memory (dynamic, | managed by runtime library
| grows up)             |
|-----|
| Uninitialized global  | from application and libraries
| variables             | placed there by linker
|-----|
| Initialized global    | from application and libraries
| variables             | placed there by linker
|-----|
| Code for library      | from application and libraries
| procedures            | placed there by linker
| ...                   |
| Code for application  |
| procedures            |
|-----| low addresses

```

Registers at program start:

*SP* = points to the initial stack top

*PC* = points to the initial code (main)

*R0* ... *Rn* = junk or zeros

#### **Runtime Library**

Typically there is a layer of code between the application and the machine or operating system: *the runtime library*.

- At start, some initialization may need to occur. The linker selects a procedure in the runtime library as the program's starting point. That procedure initializes things and then calls the application's starting point.
- There may be a special protocol to terminate a process. The runtime startup procedure can catch a return from *main()* and then do a proper termination (e.g. *syscall exit*).
- During execution the application and libraries will need to make system calls. The runtime library provides procedures that act as wrappers around the low-level system call mechanism (typically a software interrupt or trap instruction).

**Compiler support library**

High-level languages typically implement some functionality in libraries rather than having the compiler outputs a lot of code for that functionality:

- Dynamic memory management, garbage collection
- Exception handling
- Standard procedures for I/O, networking, data structures, etc