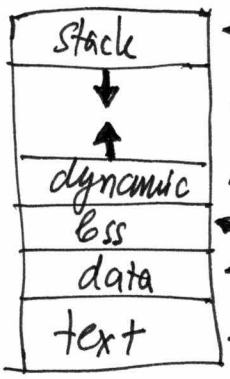
COMSW4118-3-1 9/12/2017

- Slide ("Supplemental Slides" from 9/7 Lectures), a simple program




UNIX address space (of a process)

- stack ← push/pop managed memory for function calls: arguments, local vars
- ↕ grow against each other to make optimal use of memory
- dynamic ← heap memory (malloc/free)
- bss ← uninitialized memory (set to zero) (i.e. no initial val)
- data ← stuff that is always available (globals), static (does not grow)
  - data is vars that have pre-defined values & can be modified
- text ← the program binary code

- Prefer local vars whenever you write code, then global if you know size, otherwise heap.
- System Calls
  - sole interface b/n user & kernel
  - implemented as library routines ← you call a C library function that corresponds to a system call
  - errors indicated by returns of -1; error code is errno (decode by strerror)
- fork() System Call ← same stack, same text, data, everything. (heap, etc.)
  - copies caller process and creates a child (return 0 in child & child PID in parent).
  - the two processes are completely independent
  - wait() waits for a child to complete, gets returns stat, and frees the child zombie process that has completed:

```
int pid = fork();
if(pid) { int ret_cd;
          while (pid != wait(&ret_cd)); }
```

wait(0) to ignore child return code.

waitpid() to wait for a specific child. ← still keeps it under its parent, though.

  - exec() to run a program by replacing the calling process.
    the complete image is replaced: text, data, bss, stack, heap is empty.
    several versions of exec() that allow to specify args, environment, etc.
- Each process usually sees the full memory (ex: 0 to $2^{32}$ bytes) and treats it as if it were the only program running — this is not the physical memory though, the OS uses virtual memory behind the scenes. fork() copies all pointer values, they are the same in parent & child, but the underlying physical memory will be different (although w/ same values initially).
- File descriptors: Standard: 0 = stdin, 1 = stdout, 2 = stderr
  (integer type) → read/write accept fds, open creates fds.
  printf / perror write to stdout/stderr, fprintf accepts FILE* to write to