

# Announcements

- Program #1
  - Is on the web
- Reminder about change in office hours
  - Now Tu & Th 11-12
- Reading
  - Chapter 4
  - Chapter 6 (for Tuesday)

# Project Issues

- Role of `libuser.c`

- system call routines
- `_Entry`
  - Calls Main
  - Calls Exit
- Useful functions

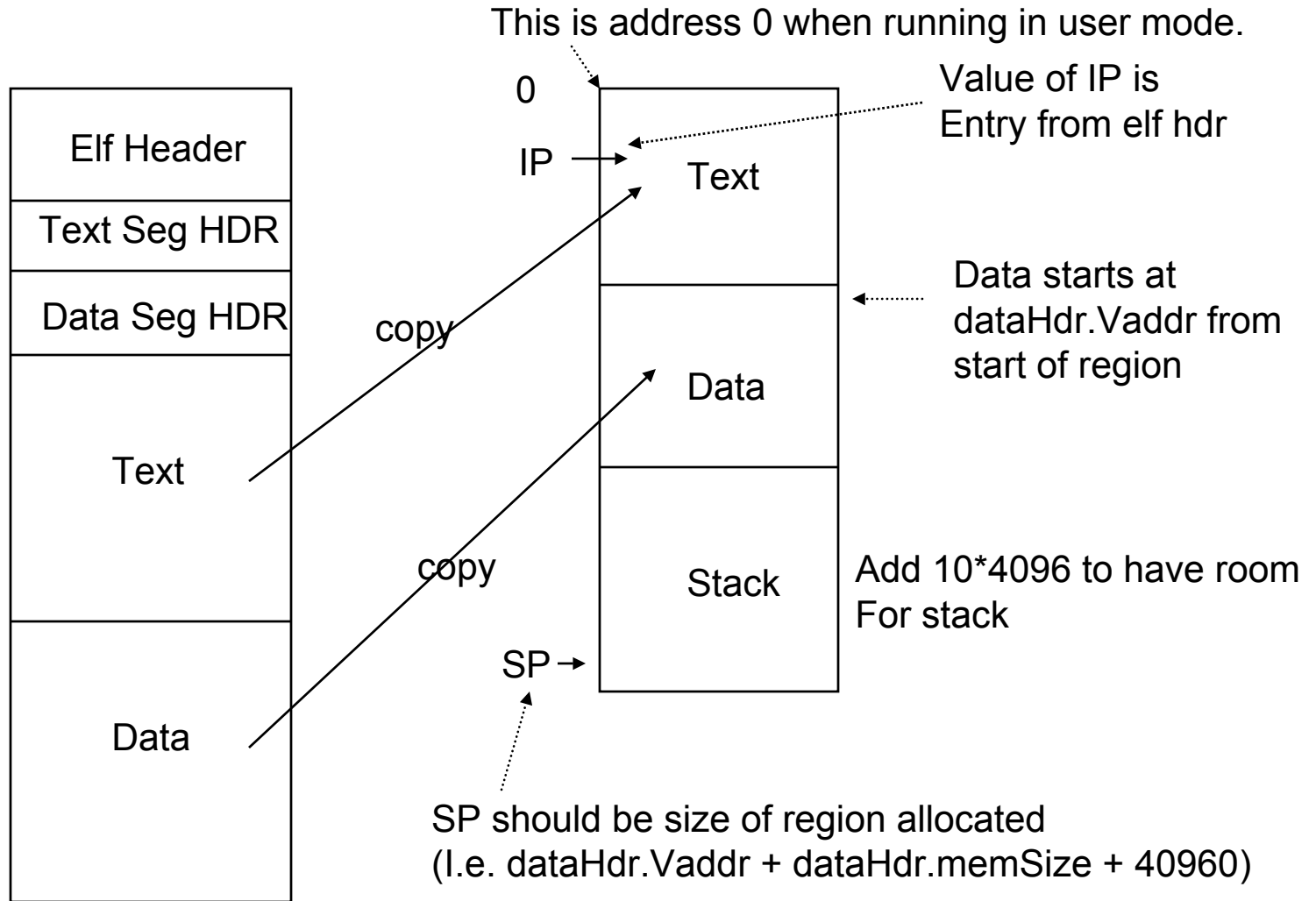
- `buildFat`

- Standalone program to build `fd.img` (floppy disk)
- Needs to contain:
  - Bootinfo
  - Kernel
  - User programs

# User Process Memory Layout

File as loaded by loadFile

User-space Process



# Process Termination

- **Process can terminate self**
  - via the exit system call
- **One process can terminate another process**
  - use the kill system call
  - can any process kill any other process?
    - No, that would be bad.
    - Normally an ancestor can terminate a descendant
- **OS kernel can terminate a process**
  - exceeds resource limits
  - tries to perform an illegal operation
- **What if a parent terminates before the child**
  - called an orphan process
  - in UNIX becomes child of the root process
  - in VMS - causes all descendants to be killed

# Termination (cont.) - UNIX example

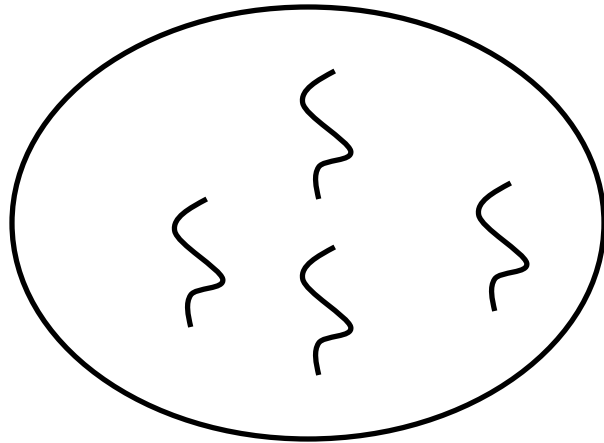
- **Kernel**
  - frees memory used by the process
  - moved process control block to the terminated queue
- **Terminated process**
  - signals parent of its death (SIGCHLD)
  - is called a zombie in UNIX
  - remains around waiting to be reclaimed
- **parent process**
  - wait system call retrieves info about the dead process
    - exit status
    - accounting information
  - signal handler is generally called the reaper
    - since its job is to collect the dead processes

# Threads

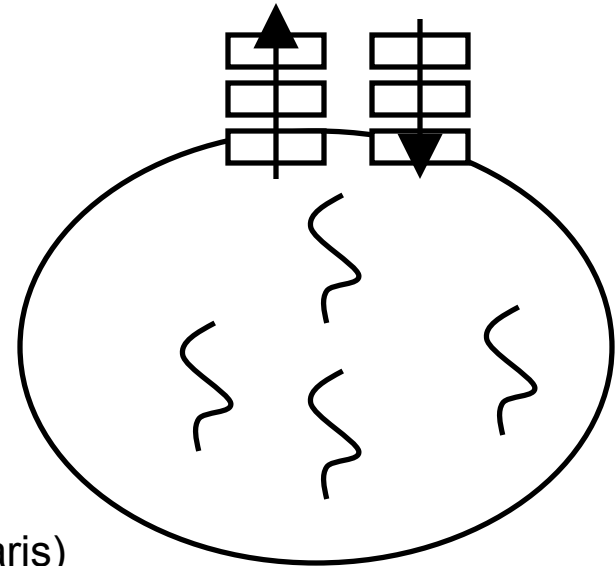
- processes can be a heavy (expensive) object
- threads are like processes but generally a collection of threads will share
  - memory (except stack)
  - open files (and buffered data)
  - signals
- can be user or system level
  - user level: kernel sees one process
    - + easy to implement by users
    - I/O management is difficult
    - in an multi-processor can't get parallelism
  - system level: kernel schedules threads

# Thread Implementation

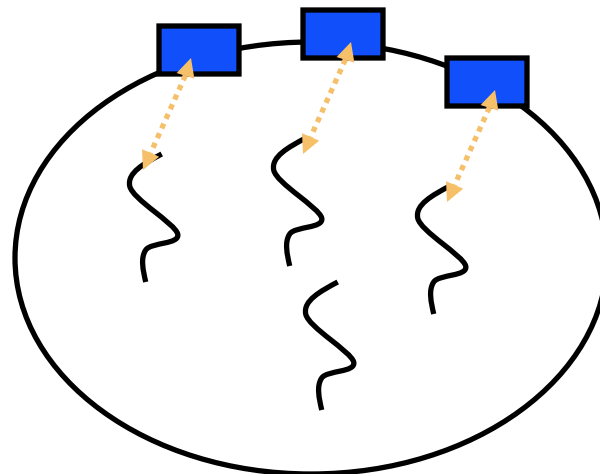
User Visible Threads



Async Kernel Calls (TruUnix 64)



Light Weight Processes (Solaris)



# Dispatcher

- The inner most part of the OS that runs processes
- Responsible for:
  - saving state into PCB when switching to a new process
  - selecting a process to run (from the ready queue)
  - loading state of another process
- Sometimes called the short term scheduler
  - but does more than schedule
- Switching between processes is called context switching
- One of the most time critical parts of the OS
- Almost never can be written completely in a high level language



# Selecting a process to run

- called scheduling
- can simply pick the first item in the queue
  - called round-robin scheduling
  - is round-robin scheduling fair?
- can use more complex schemes
  - we will study these in the future
- use alarm interrupts to switch between processes
  - when time is up, a process is put back on the end of the ready queue
  - frequency of these interrupts is an important parameter
    - typically 3-10ms on modern systems
    - need to balance overhead of switching vs. responsiveness