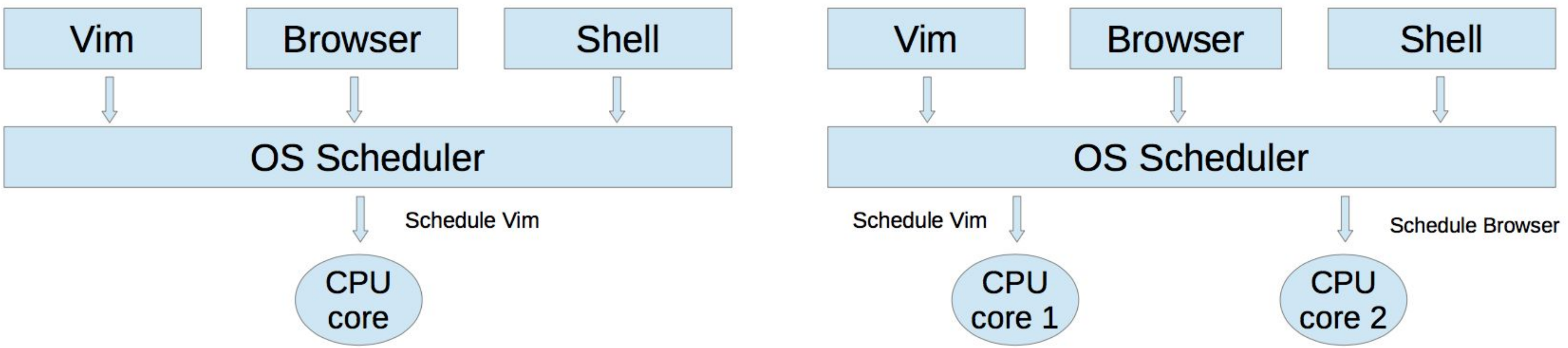
Multithreading/Parallel Processing

Week 6

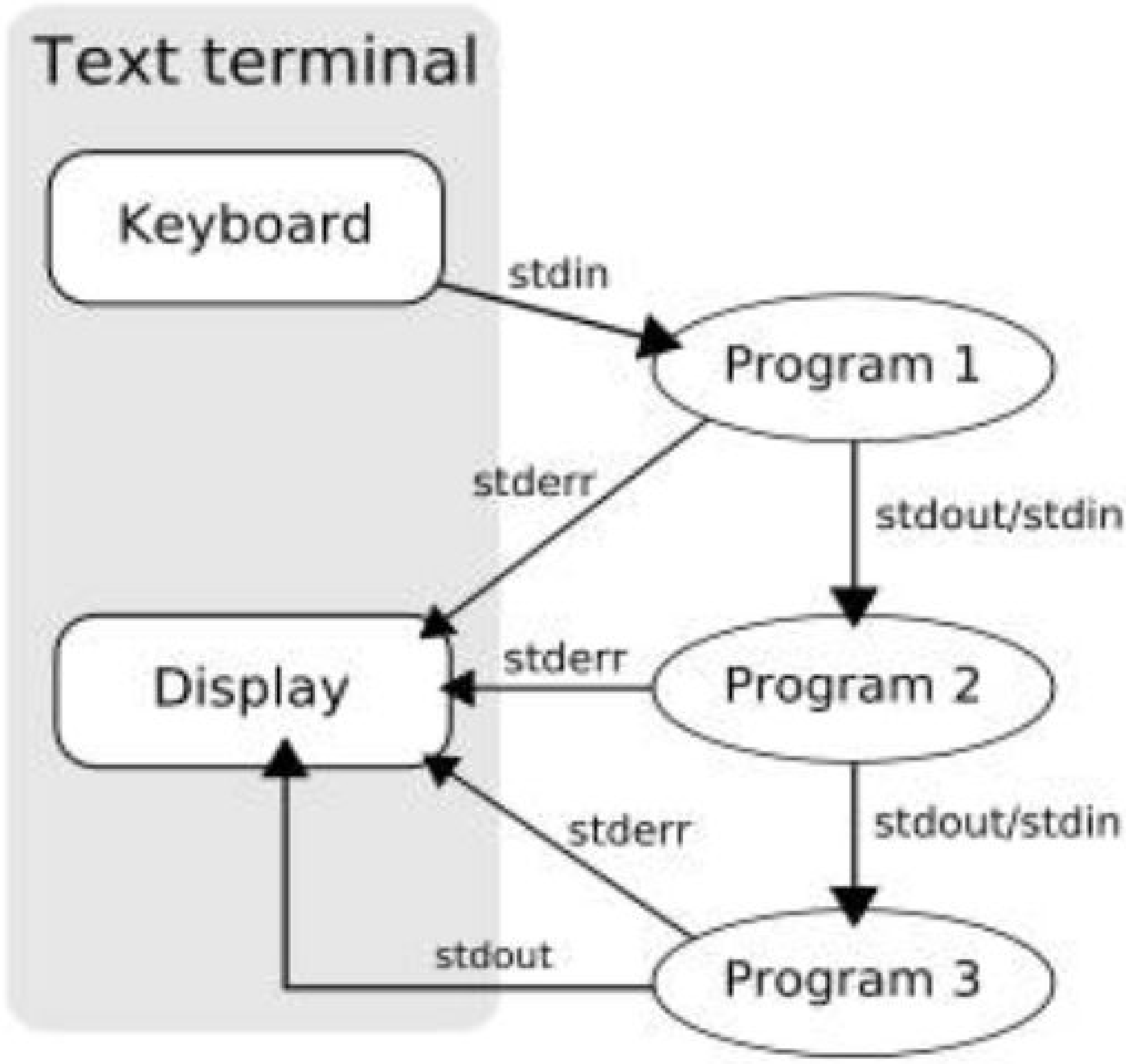
Multitasking

- Run multiple processes **simultaneously** to increase performance
- Processes do not share internal structures (stacks,globals,etc)
  - Communicate via **IPC** (inter-process communication) methods
    - Pipes, sockets, signals, message queues
- **Single core: Illusion** of parallelism by switching processes quickly (**time-sharing**). Why is illusion good?
- **Multi-core: True** parallelism. Multiple processes execute **concurrently** on different CPU cores


Vim
Browser
Shell
OS Scheduler
Schedule Vim
CPU core
Vim
Browser
Shell
OS Scheduler
Schedule Vim
Schedule Browser
CPU core 1
CPU core 2

Multitasking

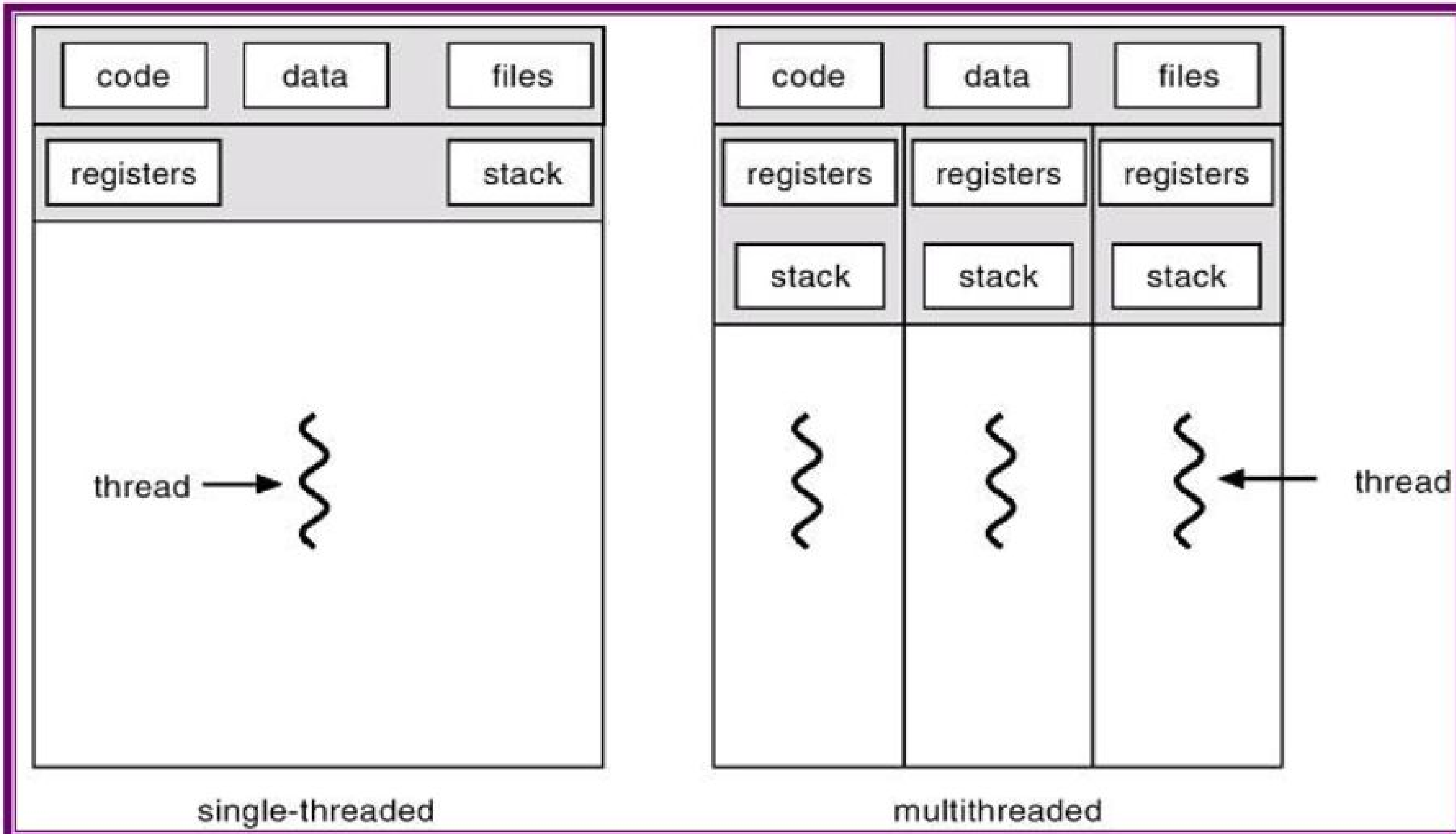
- tr -s ‘[:space:]’ ‘\n’ | sort -u | comm -23 - words
- Three separate processes spawned simultaneously
  - P1 - tr
  - P2 - sort
  - P3 - comm
- Common buffers (pipes) exist between 2 processes for communication
  - ‘tr’ writes its stdout to a buffer that is read by ‘sort’
  - ‘sort’ can execute, as and when data is available in the buffer
  - Similarly, a buffer is used for communicating between ‘sort’ and ‘comm


Text terminal
Keyboard
stdin
Program 1
stderr
stdout/stdin
Display
stderr
Program 2
stdout/stdin
stderr
stdout
Program 3

Threads

- A flow of instructions, path of execution within a process
- It is a basic unit of CPU utilization
- Each thread has its own:
  - Stack
  - Registers
  - Thread ID
- Each thread shares the following with other threads belonging to the same process
  - Code
  - Heap
  - Global Data
  - OS resources (files,I/O)
- A process can be single-threaded or multi-threaded
- Threads in a process can run in parallel

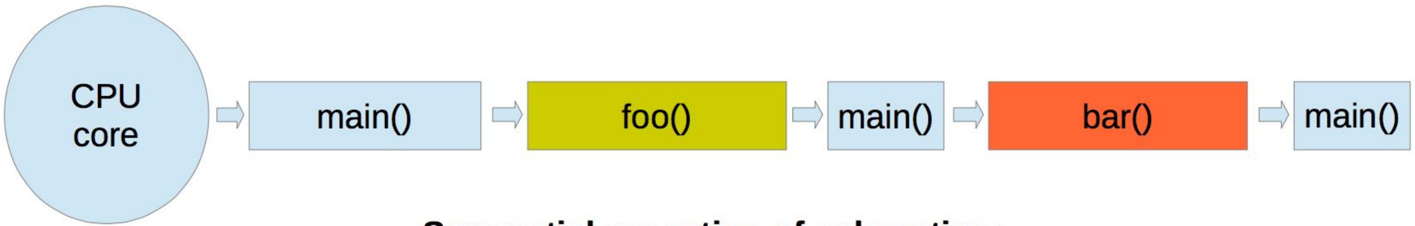
  (provide another type of parallelism)


code
data
files
registers
stack
thread
single-threaded
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded

Single threaded execution

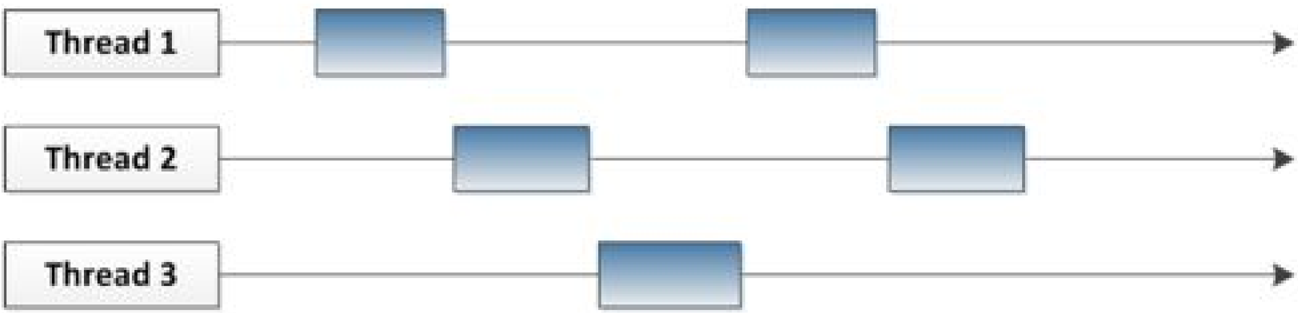
```
int global_counter = 0
int main()
{
      …
      foo(arg1,arg2);
      bar(arg3,arg4,arg5);
      …
      return 0;
}
```

```
void foo(arg1,arg2)
{
      //code for foo
}
void bar(arg3,arg4,arg5)
{
      //code for bar
}
```



Sequential execution of subroutines

Multiple threads sharing a single CPU

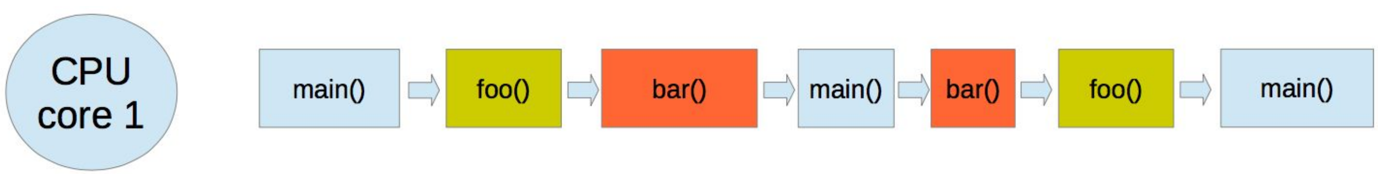


Multiple threads on multiple CPUs

Multi threaded execution (single core)

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```

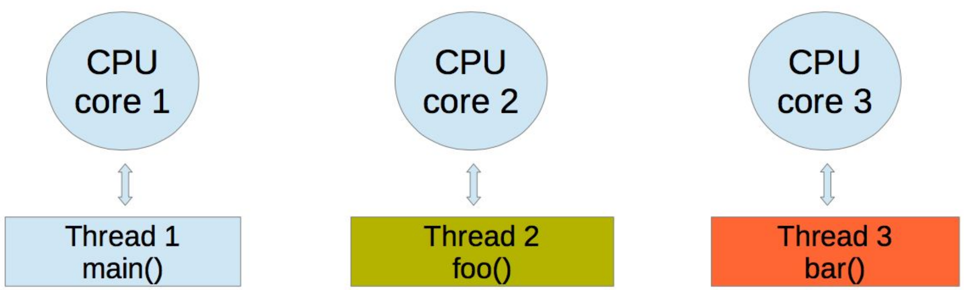


Time Sharing – Illusion of multithreaded parallelism
(Thread switching has less overhead compared to process switching)

Multi threaded execution (multiple cores)

```
int global_counter = 0
int main()
{
    …
    foo(arg1,arg2);
    bar(arg3,arg4,arg5);
    …
    return 0;
}
```

```
void foo(arg1,arg2)
{
    //code for foo
}
void bar(arg3,arg4,arg5)
{
    //code for bar
}
```



True multithreaded parallelism

Multithreading properties

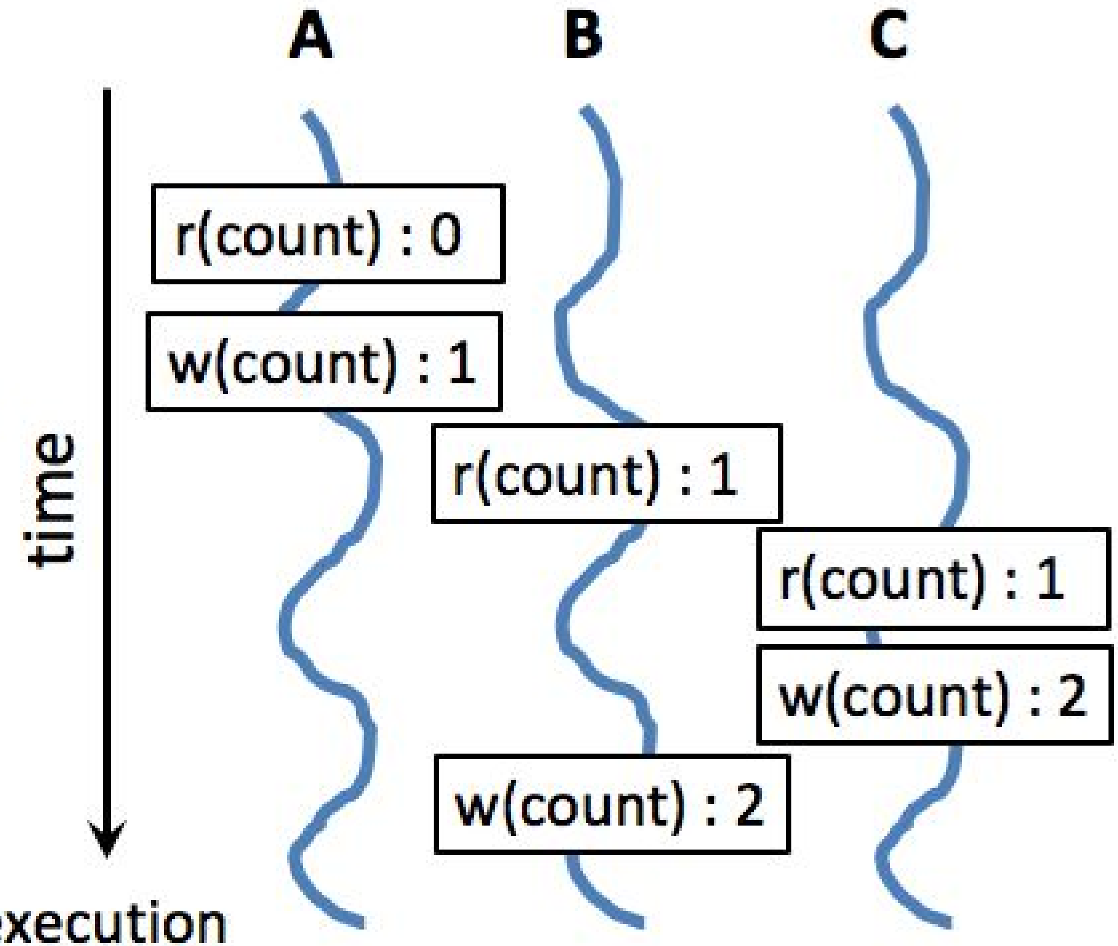
- Efficient way to **parallelize** tasks
- **Thread switches are less expensive** compared to process switches (context switching)
- Inter-thread communication is easy, via **shared global** data (heap)
- Need **synchronization** among threads accessing same data

Shared Memory

- Makes multithreaded programming
  - Powerful
    can easily access data and share it among threads
  - More efficient
    No need for system calls when sharing data
    Thread creation and destruction less expensive than process creation and destruction
  - Non-trivial
    Have to prevent several threads from accessing and changing the same shared data at the same time (synchronization)

Race Condition

```
int count = 0;
void increment()
{
  count = count + 1;
}
```



Result depends on order of execution

=> Synchronization needed

Lab 6

- Evaluate the performance of multithreaded ‘sort’ command
  - **od -An -f -N 4000000 < /dev/urandom | tr -s ' ' '\n' > random.txt**
  - Might have to modify the command above
- Delete the empty line
  - **time -p sort -g --parallel=2 numbers.txt > /dev/null**
- Add /usr/local/cs/bin to PATH
  - $ export PATH=/usr/local/cs/bin:$PATH
- Generate a file containing 10M random **single-precision floating point numbers**, one per line with no white space
  - /dev/urandom: pseudo-random number generator

Lab 6

- od
  - write the contents of its input files to standard output in a user-specified format
  - Options
    - -t f: Double-precision floating point
    - -N <count>: Format no more than *count* bytes of input
- sed, tr
  - Remove address, delete spaces, add newlines between each float

Lab 6

- use time -p to time the command sort -g on the data you generated
- Send output to /dev/null
- Run sort with the --parallel option and the
  - –g option: compare by general numeric value
  - Use time command to record the real, user and system time when running sort with 1, 2, 4, and 8 threads
    - $ time –p sort –g file_name > /dev/null (1 thread)
    - $ time –p sort –g --parallel=[2, 4, or 8] file_name > /dev/null
  - Record the times and steps in log.txt