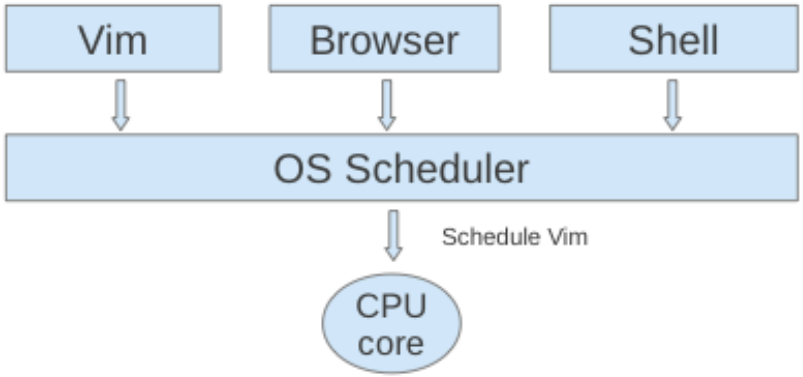
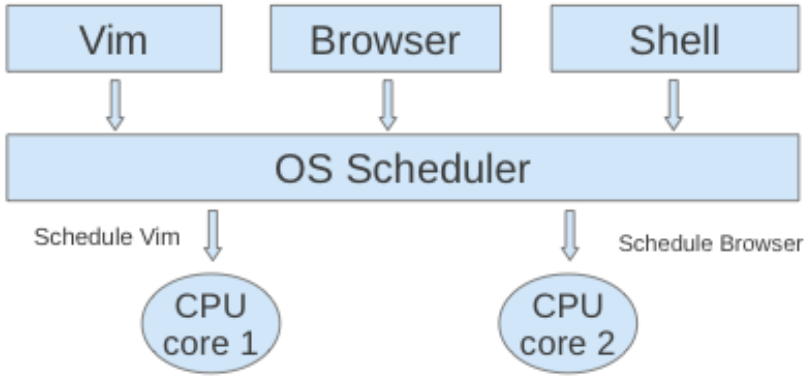
# Multithreaded Performance

Week 9 - Monday

# Multitasking

- Run multiple processes simultaneously to increase performance
- Processes do not share internal structures(stack, variable)
  - Communicate via IPC(inter-process communication) methods
    - Pipes,sockets, signals ,etc.
- Single core: Illusion of parallelism
- Multi-core: True parallelism



# 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 when data is available in the buffer
  - Similarly, a buffer is used for communicating between ‘sort’ and ‘comm’

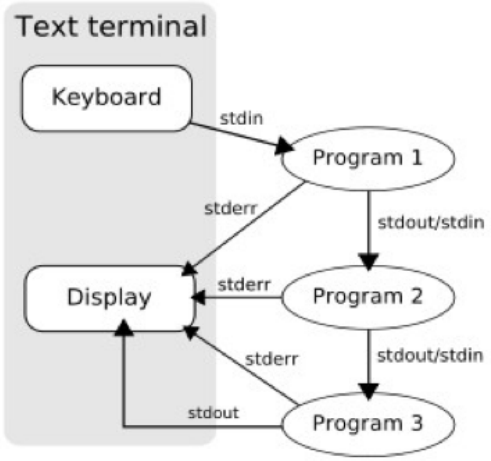


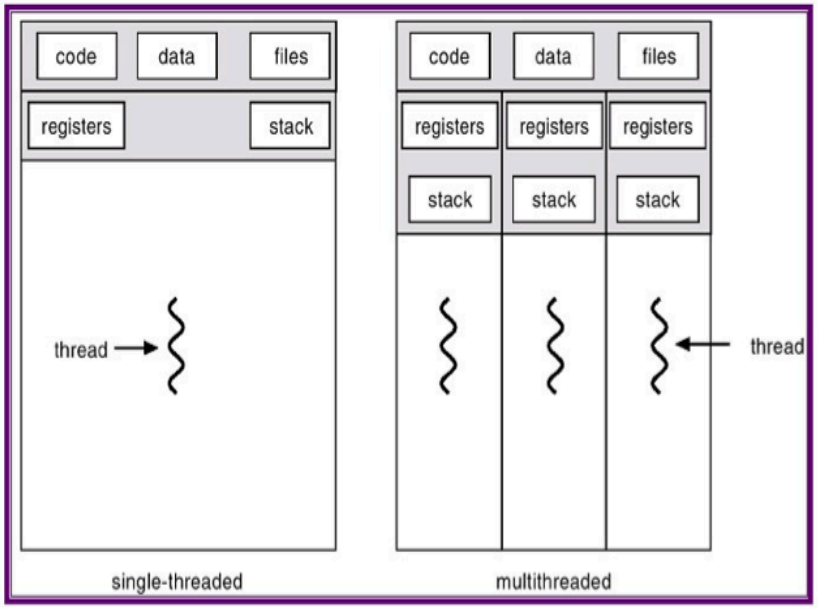

Image source: Wikipedia

# Thread vs Process

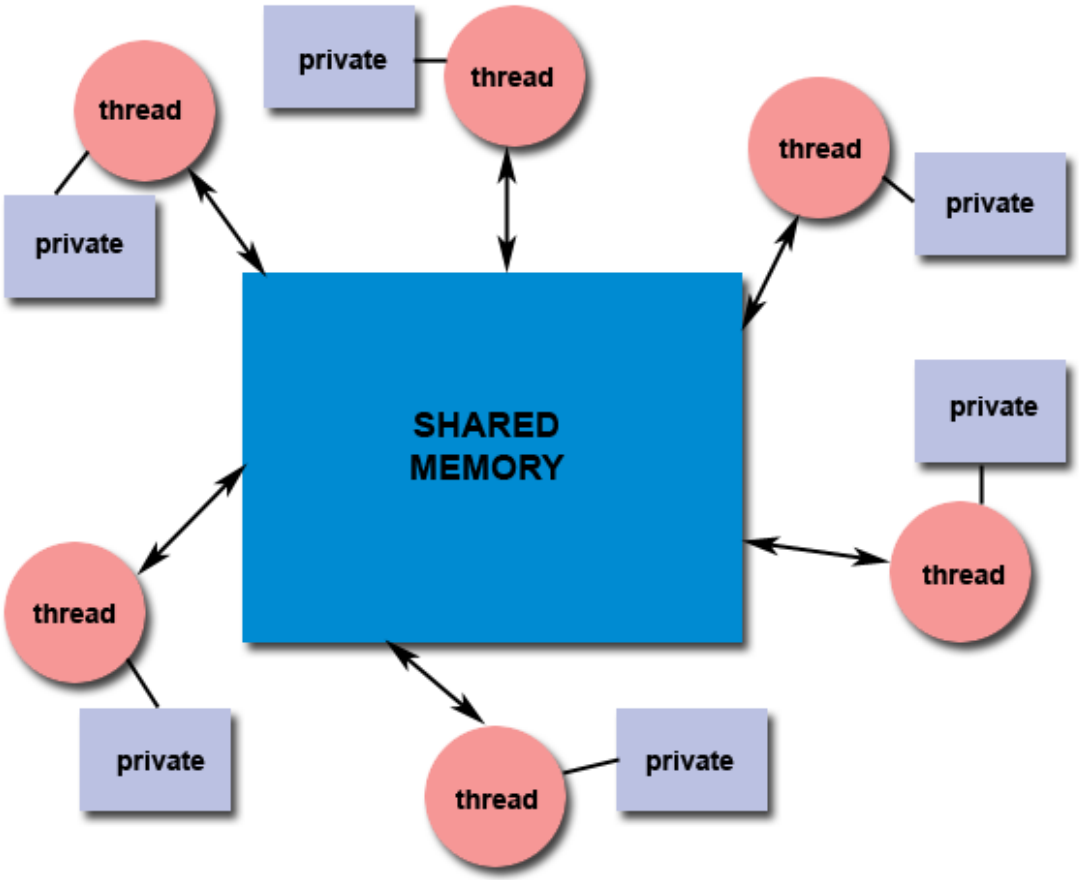
- **Process**: An executing instance of an application
- **Thread**: Path of execution within a process
- A process can contain multiple threads
- When start an application, the OS creates a process and begins executing the primary thread of that process
- Note: Threads within the same process share the same address space, whereas different processes do not.
- This allows threads to read from and write to the same data structures and variables

# Threads

- Memory
  - Private v/s Shared
- Each thread has its own
  - Stack
  - Registers
  - Thread ID
- Each thread shares the following with other threads in same process
  - Code
  - Global data
  - OS resources (files, I/O)
- Communication always through shared memory

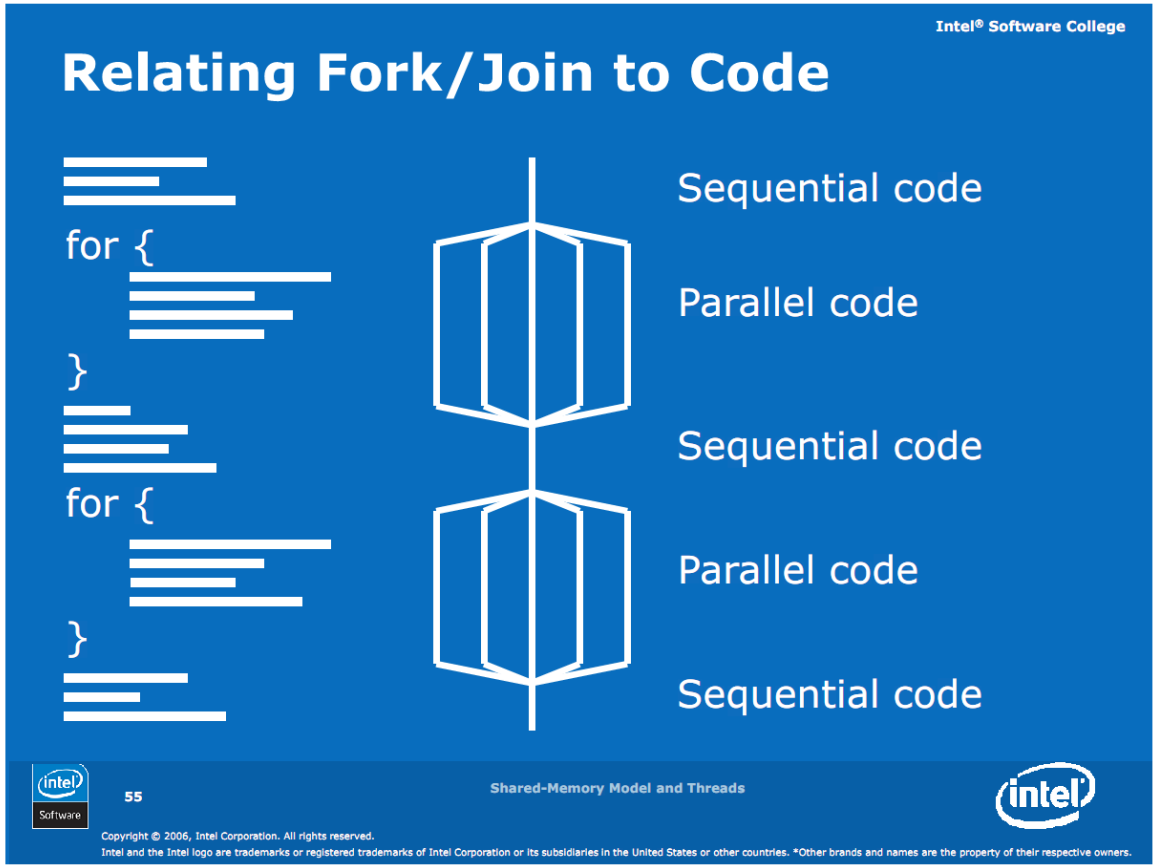


# Shared Memory Model



# Threads

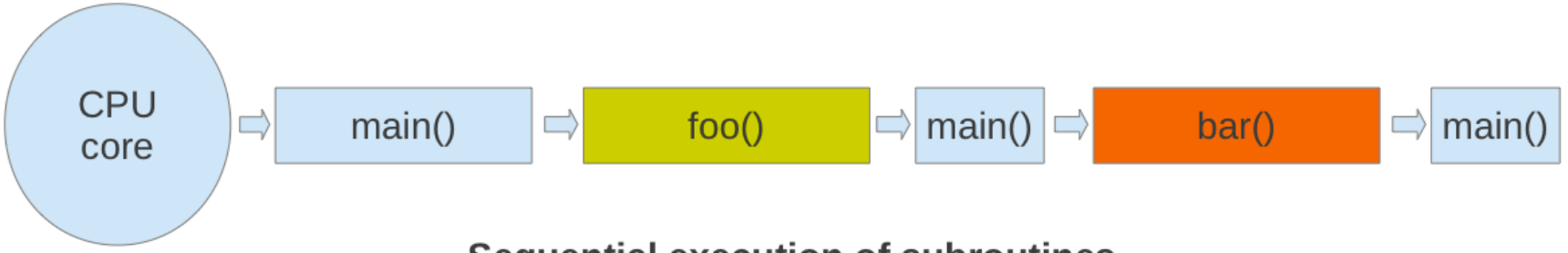
- Most common programming model for threads
  - Fork-join model



# Single threaded execution

```
int globalCounter = 0;

int main() {
...
foo(arg1, arg2);
bar(arg3, arg4, arg5);
...
return 0;

}
```

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

void foo(arg1, arg2) {
//code for foo
}

}
```

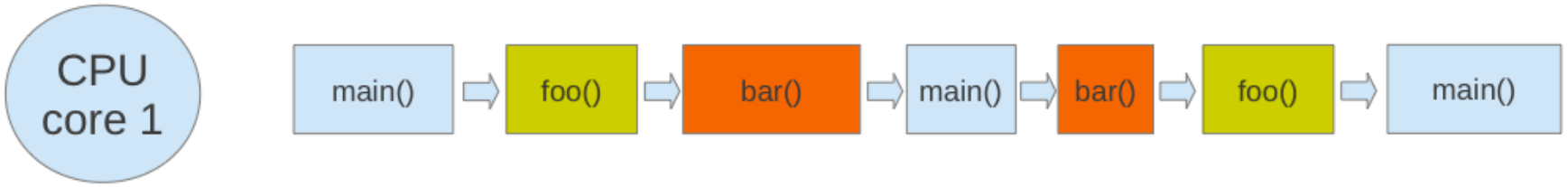



Sequential execution of subroutines

# Multi-threaded (single core)

```
int globalCounter = 0;

int main() {
...
Run thread foo(arg1, arg2);
Run thread bar(arg3, arg4, arg5);
...
return 0;

}
```

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

void foo(arg1, arg2) {
//code for foo
}

}
```

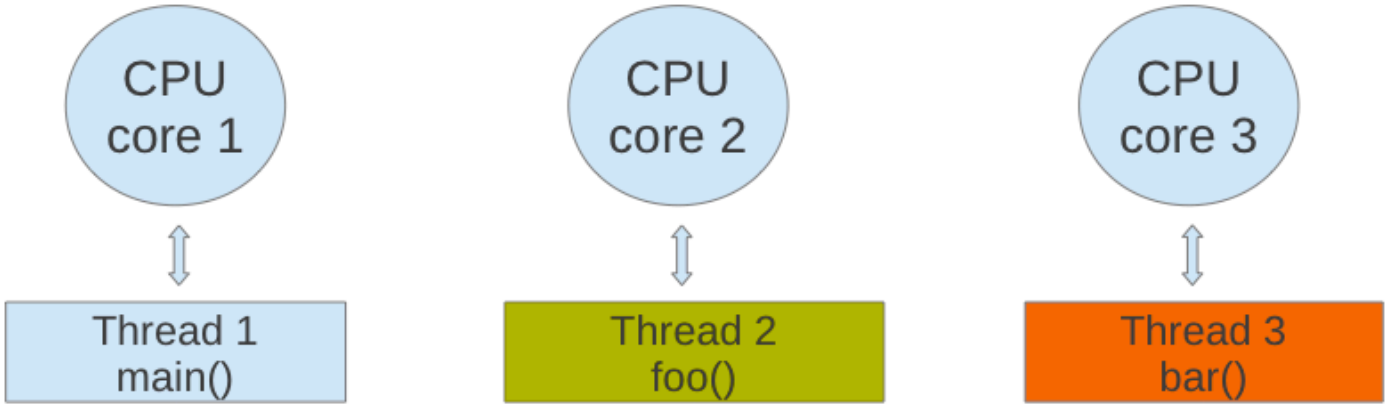



**Time Sharing – Illusion of multithreaded parallelism**
**(Thread switching has less overhead compared to process switching)**

# Multi-threaded (multi core)

```
int globalCounter = 0;

int main() {
...
Run thread foo(arg1, arg2);
Run thread bar(arg3, arg4, arg5);
...
return 0;
}
```

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

void foo(arg1, arg2) {
//code for foo
}

}
```




**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
- Need synchronization among threads accessing same data

# PThread API

#include <pthread.h>

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void* (*thread_function)(void *), void *arg);
  - Returns 0 on success, otherwise returns non-zero error number
- void pthread_exit(void *retval);
- int pthread_join(pthread_t thread, void **retval);
  - Returns 0 on success, otherwise returns non-zero error number

# PThread example

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<unistd.h>

void* thread_fun(void *arg) {
    int thread_num = *(int *)arg;
    for (int i = 0; i < 5; i ++) {
        printf("Thread #%d: count %d\n",
                thread_num, i);
        sleep(1);
    }
    printf("Thread #%d exit\n", thread_num);
    pthread_exit(NULL);
}
```

Gcc with –lpthread or -pthread!

```
int main() {
    pthread_t t1, t2;
    int *n1, *n2;
    n1 = malloc(sizeof(int));
    *n1 = 1;
    n2 = malloc(sizeof(int));
    *n2 = 2;
    printf("Create Thread #1\n");
    pthread_create(&t1, NULL,
            thread_fun, (void *)n1);
    printf("Create Thread #2\n");
    pthread_create(&t2, NULL,
            thread_fun, (void *)n2);
    printf("Finish creating threads\n");
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    printf("Both threads have exited\n");
    return 0;
}
```

# Thread Synchronization

- A simple example
  - A thread wait for another thread
  - pthread_join()
- The reality is much more complicated
  - For example, read/write shared data
  - Solution: all kinds of locks

# Lab

- Environment variable for the right *sort*
  - Export PATH=/usr/local/cs/bin:$PATH
- *od* to read random bytes
  - -A: how file offset are printed
  - -t: output format (translate bytes into different types)
  - -N: limit the # of bytes to read
- *tr/sed* to format the output
  - Replace space with newline
- Only *time* the *sort* program
  - Write the input of *sort* into a file
  - Write the output of *sort* into /dev/null

# Homework

- Ray-Tracing: Powerful rendering technique in Computer Graphics
- Yields very high quality rendering
  - Suited for scenes with complex light interactions
  - Visually realistic
  - Trace the path of light in the scene
- Computationally very expensive
  - Not suited for rendering in real-time (example:games)
  - Suited for rendering high-quality pictures
- Embarrassingly parallel
  - Good candidate for multi-threading
  - Threads need not synchronize with each other, because each thread works on a different pixel