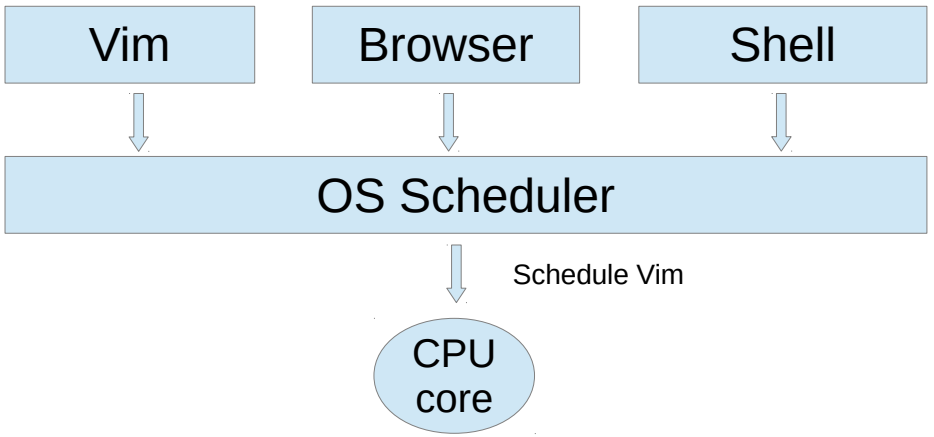
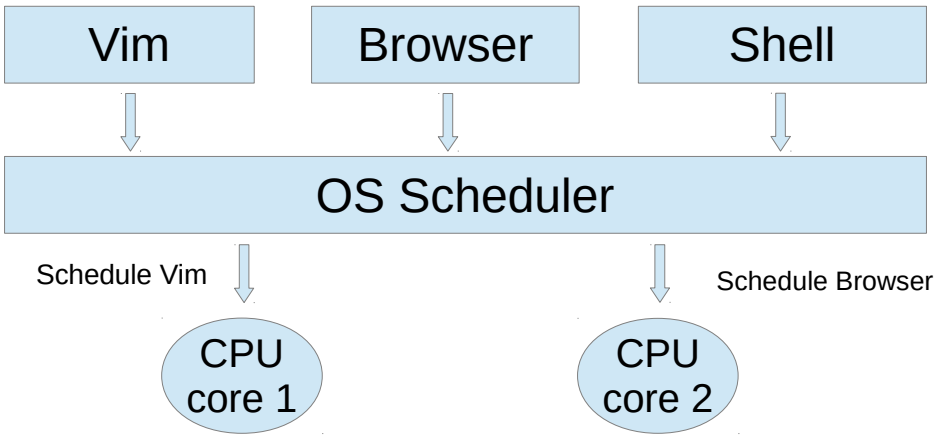
Multithreading/Parallel Processing

Week 8

Multitasking

- Run multiple processes **simultaneously** to increase performance
- Processes do not share internal structures (stack, globals, etc)
  - Communicate via **IPC** (inter-process communication) methods
    - Pipes, Sockets, Signals, Message Queues, etc
- **Single-core: Illusion** of parallelism by switching processes quickly (**time-sharing**)
- **Multi-core: True** parallelism. Multiple processes execute **concurrently** on different CPU cores





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'

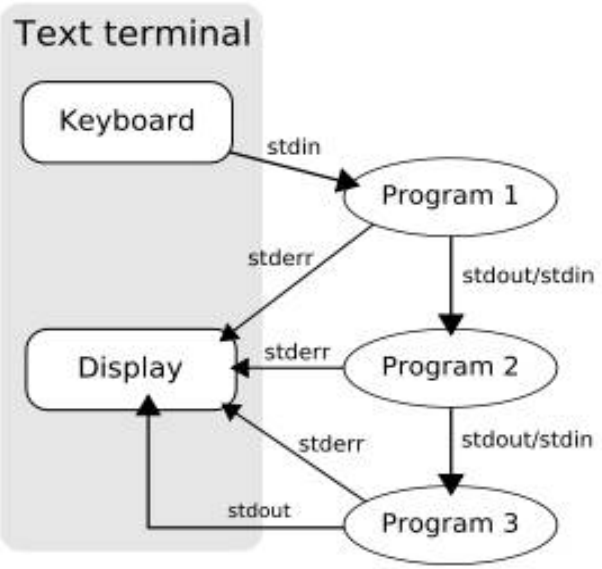
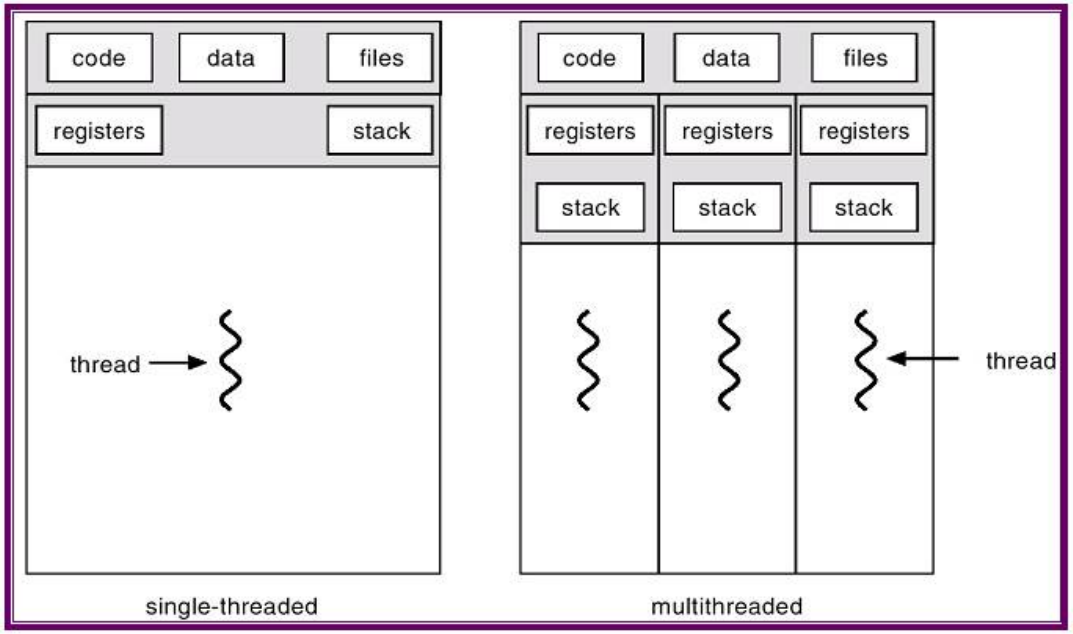


Image source: Wikipedia

Threads

- A process can be
  - **Single-threaded**
  - **Multi-threaded**
- Threads in a process can run in **parallel**
- A thread is a **lightweight 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
  - Global Data
  - OS resources (files, I/O)

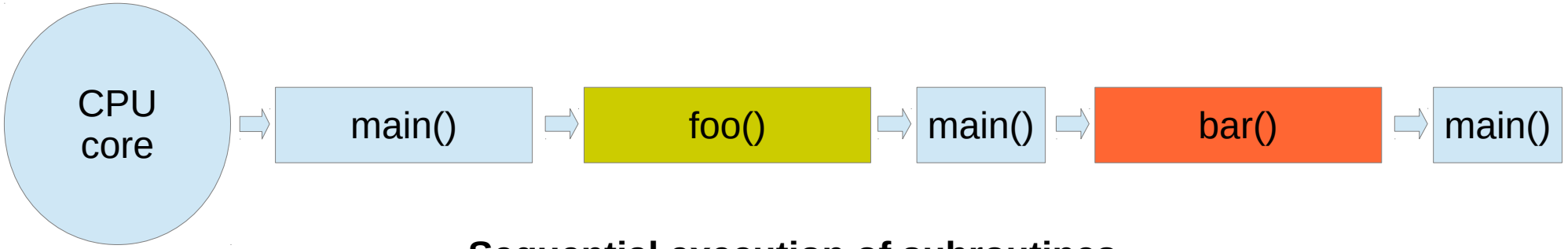


Single threaded execution

```
int globalCounter = 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

Multi-threaded execution (single core)

- int globalCounter = 0;

  int **main**()

  {

  …

  Run thread foo(arg1, arg2);

  Run thread bar(arg3, arg4, arg5);

  …

  return 0;

  }

- void **foo**(arg1, arg2)

  {

  //code for foo

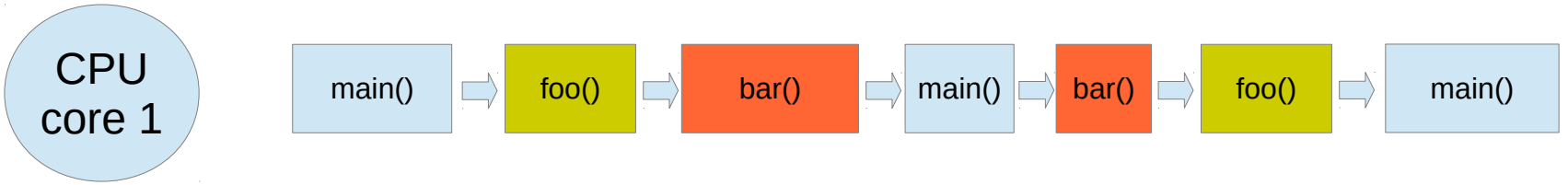
  }

- 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

- int globalCounter = 0;

  int **main**()

  {

  …

  Run thread foo(arg1, arg2);

  Run thread bar(arg3, arg4, arg5);

  …

  return 0;

  }

- void **foo**(arg1, arg2)

  {

  //code for foo

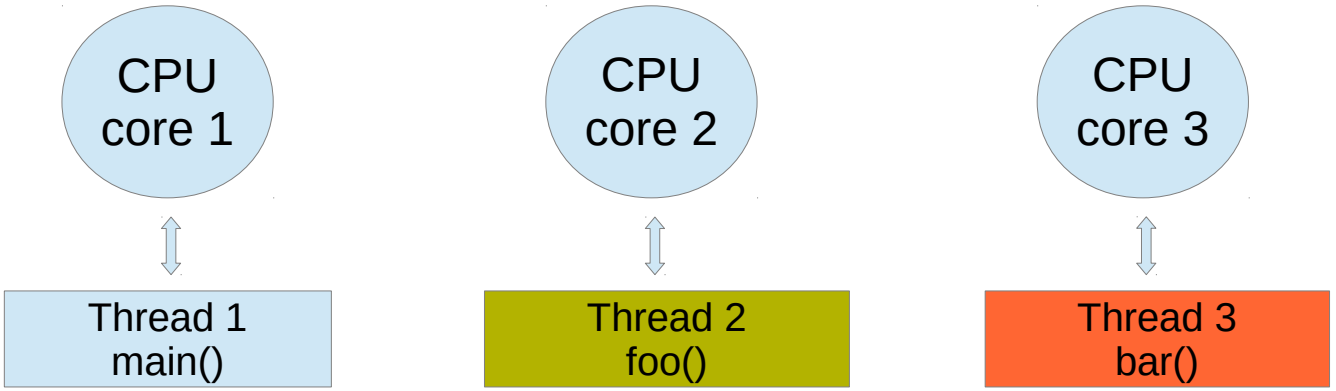
  }

- void **bar**(arg3, arg4, arg5)

  {

  //code for bar

  }



True multithreaded parallelism

Multithreading properties

- Efficient way to **parallellize** 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
  ```

Pthreads sample code

Compile the following code as – **gcc main.c -lpthread**

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

void* ThreadFunction(void *arg)
{
  long tID = (long)arg;
  printf("Inside thread function with ID = %ld\n", tID);
  pthread_exit(0);
}

int main(int argc, char *argv[])
{
  const int nthreads = 5;
  pthread_t threadID[nthreads];
  long t;
  for(t = 0; t < nthreads; ++t) {
    int rs = pthread_create(&threadID[t], 0, ThreadFunction, (void*)t);
    if(rs) {
      fprintf(stderr, "Error creating thread\n");
      return -1;
    }
  }

  printf("Main thread finished creating threads\n");

  for(t = 0; t < nthreads; ++t) {
    void *retVal;
    int rs = pthread_join(threadID[t], &retVal);
    if(rs) {
      fprintf(stderr, "Error joining thread\n");
      return -1;
    }
  }

  printf("Main thread finished execution!\n");

  return 0;
}
```

Thread safety/synchronization

- **Thread safe function –** Safe to be called by multiple threads at the same time. Function is free of 'race conditions' when called by multiple threads simultaneously.
- **Race condition –** The output depends on the order of execution
  - Shared data changed by 2 threads
    - int balance = 1000;
  - Thread 1
    - T1 - Read balance
    - T1 - Deduct 50 from balance
    - T1 - Update balance with new value
  - Thread 2
    - T2 - Read balance
    - T2 - Add 150 to balance
    - T2 - Update balance with new value

Thread synchronization

- **Order 1**
  - balance = 1000
  - T1 – Read balance (1000)
  - T1 – Deduct 50
    - 950 in temporary result
  - T2 – Read balance (1000)
  - T1 – Update balance
    - balance is 950 at this point
  - T2 – Add 150 to balance
    - 1150 in temporary result
  - T2 – Update balance
    - balance is 1150 at this point
- **The final value of balance is 1150**

- **Order 2**
  - balance = 1000
  - T1 – Read balance (1000)
  - T2 – Read balance (1000)
  - T2 – Add 150 to balance
    - 1150 in temporary result
  - T1 – Deduct 50
    - 950 in temporary result
  - T2 – Update balance
    - balance is 1150 at this point
  - T1 – Update balance
    - balance is 950 at this point
- **The final value of balance is 950**

Thread Synchronization

- **Mutex (Mutual exclusion)**
  - Thread 1

    Mutex.lock()

    Read balance

    Deduct 50 from balance

    Update balance with new value

    Mutex.unlock()
  - Thread 2

    Mutex.lock()

    Read balance

    Add 150 to balance

    Update balance with new value

    Mutex.unlock()
  - balance = 1100
- Only one thread will get the mutex. Other threads will **block in Mutex.lock()**.
- Other threads can start execution only when the thread having the mutex calls **Mutex.unlock()**

Lab

- Evaluate performance of multithreaded 'sort' command
- `od -An -f -N 4000000 < /dev/urandom | tr -s ' ' '\n' > random.txt`
- You might have to modify the above command
- Delete empty line
- `time -p sort -g --parallel=2 numbers.txt > /dev/null`

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

Ray-tracing

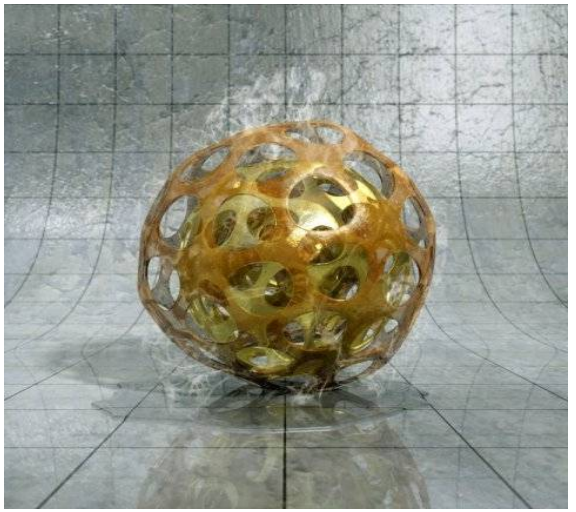
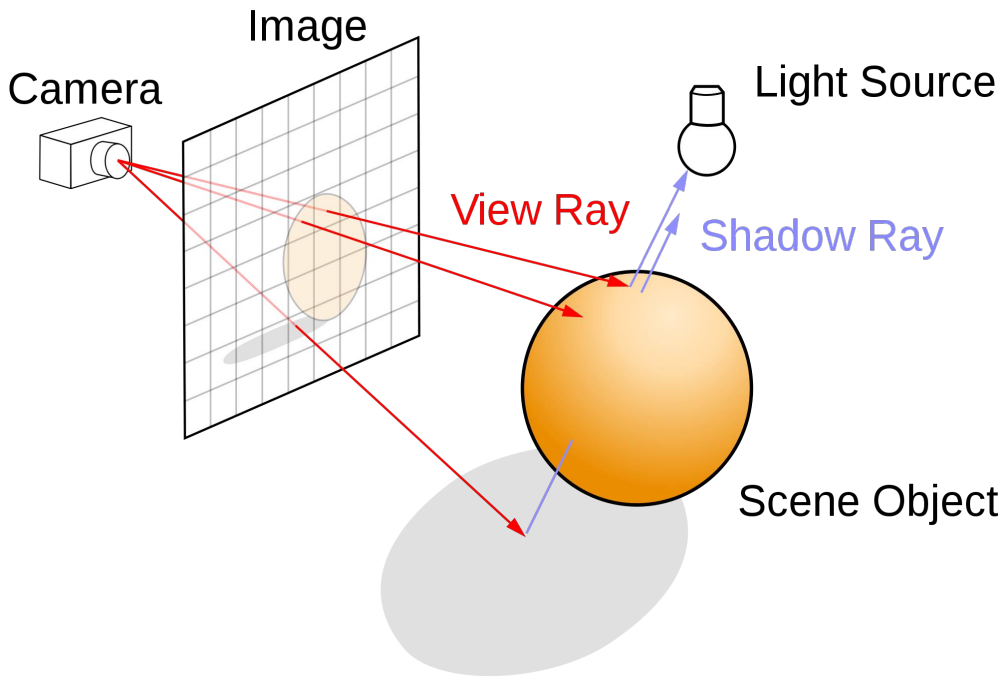


Image Source: POV Ray, Hall of Fame

Ray-tracing

- Trace the path of a ray from the eye
  - **One ray per pixel** in the view window
  - The color of the ray is the color of the corresponding pixel
- Check for **intersection** of ray with scene objects.
- **Lighting**
  - **Flat shading** – The whole object has uniform brightness
  - **Lambertian shading** – Cosine of angle between surface normal and light direction



Homework Output

Homework - Anti-Aliasing