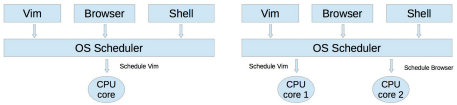
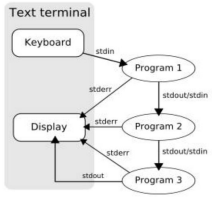
Multitasking

- 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



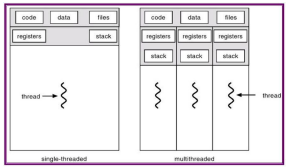
Multitasking

- `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



Threads

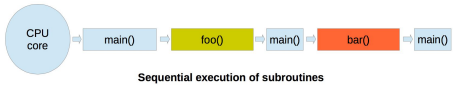
- A process can be
  - Single-threaded
  - Multi-threaded
- Threads in a process can run in parallel
- A thread is a lightweight process
- It is a basic unit of CPU utilization
- Each thread has its own:
  - Stack
  - Registers
  - Thread ID
- Each thread shares the following with other threads belonging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



Single threaded execution

```
int 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
}
```

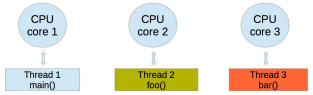


Sequential execution of subroutines

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
}
```

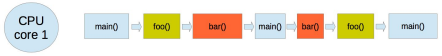


True multithreaded parallelism

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)

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 number
- `void pthread_exit(void *retval);`
- `int pthread_join(pthread_t thread, void **retval);`
  - Returns 0 on success, otherwise returns non zero error number

```
#include<pthread.h>  //Compile the following code as - gcc main.c -lpthread
#include<stdio.h>

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

void CreateThreads(const int nthreads,
                   pthread_t* threadID) {
    for(long t = 0; t < nthreads; ++t) {
        int rs = pthread_create(&threadID[t], 0,
                    ThreadFunction, (void*)t);
        if(rs)
            fprintf(stderr,"Creation error\n");
            exit(1);
        }
    }
    printf("Finished creating threads\n");
}
```

```
void JoinThreads(const int nthreads,
                 pthread_t* threadID) {
    for(long t = 0; t < nthreads; ++t) {
        void *retVal;
        int rs = pthread_join(threadID[t],
                              &retVal);
        if(rs) {
            fprintf(stderr, "Joining error\n");
            exit(1);
        }
    }
}

int main(int argc, char *argv[]) {
    const int nthreads = 5;
    pthread_t threadID[nthreads];

    CreateThreads(nthreads, threadID);
    JoinThreads(nthreads, threadID);

    printf("Main thread finished execution!\n");
    return 0;
}
```

Pthread API

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 safety/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 thread will **block in Mutex.lock()**
- Other thread can start execution only when the thread that holds the mutex calls **Mutex.unlock()**

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 **double-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

Ray-Tracing

- **Powerful rendering technique in Computer Graphics**
- **Yields high quality rendering**
  - Suited for scences with complex light interactions
  - Visually realistic for a wider variety of materials
  - Trace the path of light in the scene
- **Computationally expensive**
  - Not suited for real-time rendering (e.g. games)
  - Suited for rendering high quality pictures (e.g. movies)
- **Embarrassingly parallel**
  - Good candidate for **multi-threading**
  - Threads need **not synchronize** with each other, because each thread works on a different pixel (at least at small scale)