CSC4005 Parallel Programming Tutorial 4

Bokai Xu, 119010355@link.cuhk.edu.cn

Outline of Tutorial 4

- **Process model**
  - What is a process
  - How do OS represent a process?
  - Process Address Space
  - State of a process
  - Context Switch
  - Type of a process
- **Thread model**
  - Introduction to thread
  - Web Server Example
  - Multithreading
- **Pthread API**
  - Creating Threads
  - Terminating Threads
  - Synchronization
  - Mutual Exclusive Lock
  - Signal & Condition Variable
- **Project 2**

What is a process

A program is static file.

A process is **an** **instance** **of a program** that is being executed. (dynamic)

A single program can create multiple processes.

Features of Process

- **Each process exists within its** **own address or memory space**.
- Each process **is independent and treated as an isolated process by the OS.**
- Processes need Inter-process Communication in order to communicate with each other.

Acknowledgement: Lele Li (TA of CSC3150)

Process-thread model

A method to **model the behavior** of a program running on a CPU core.

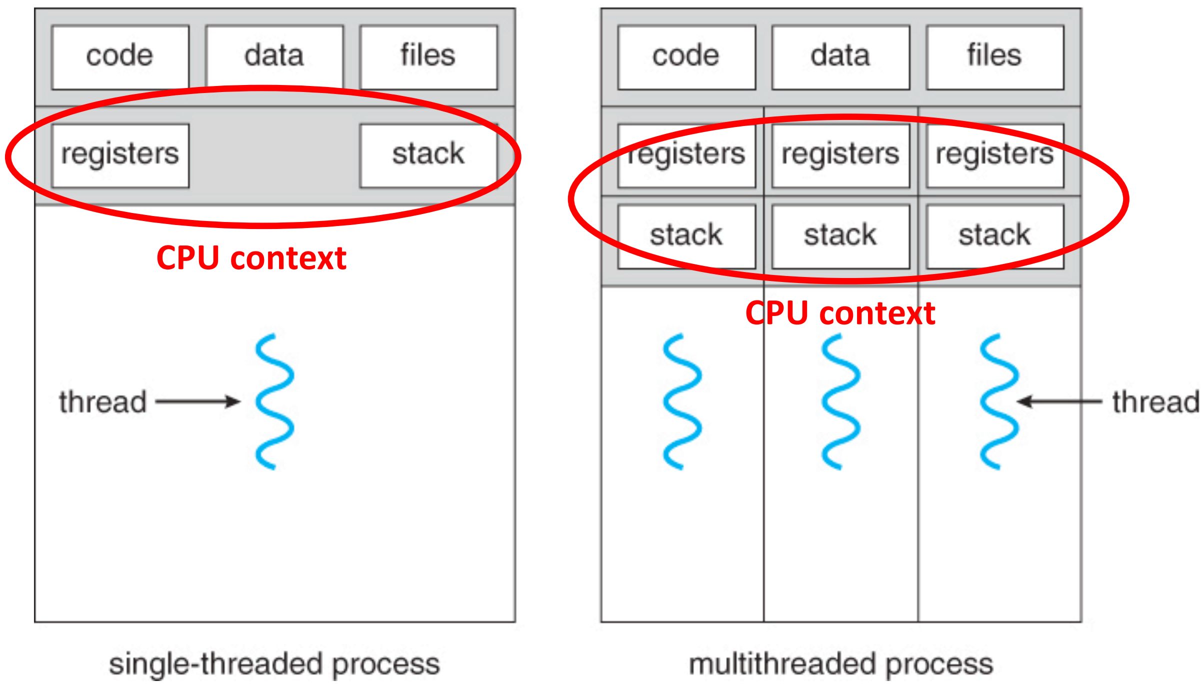
The behavior of **multiple** instances of **multiple** programs on **multiple** cpu cores.

Earliest:

Only process model

Later:

Process-thread model



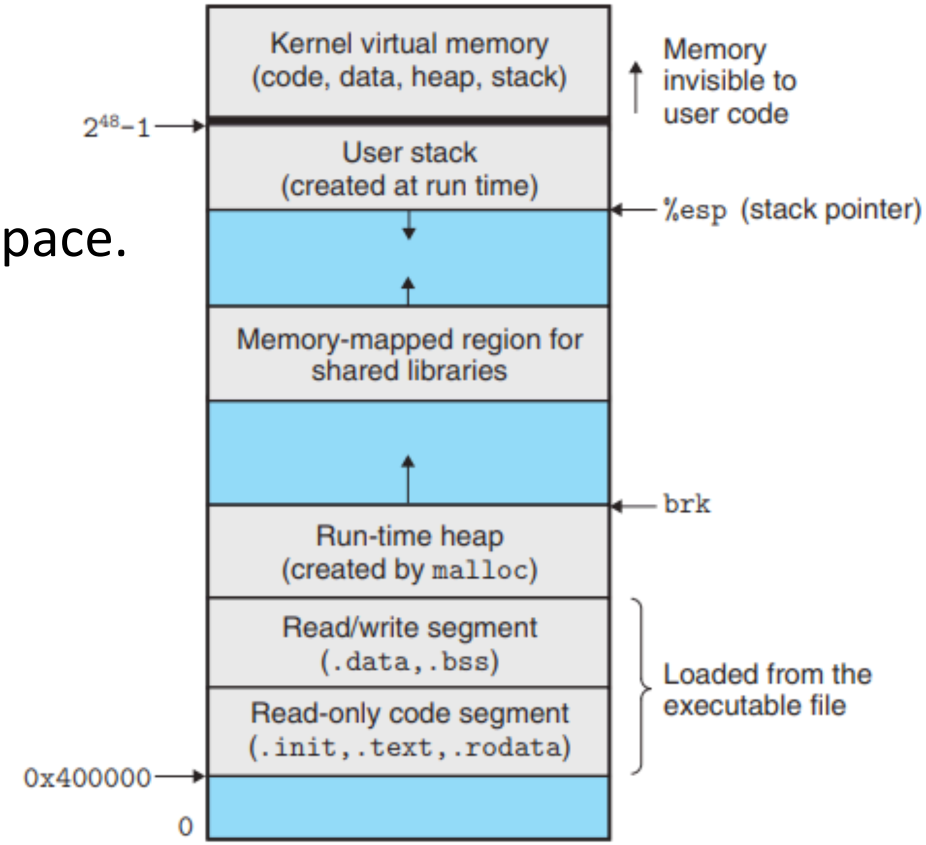
How do OS represent a process?

- Process Control Block (PCB)
- A data structure to describe a process.
- A process maps an unique PCB.
- A PCB Contains:
  - Resource (memory, files opened)
  - CPU context (registers, program counter, pointers, .. **CPU context (CPU现场)**)
  - …

Process Address Space

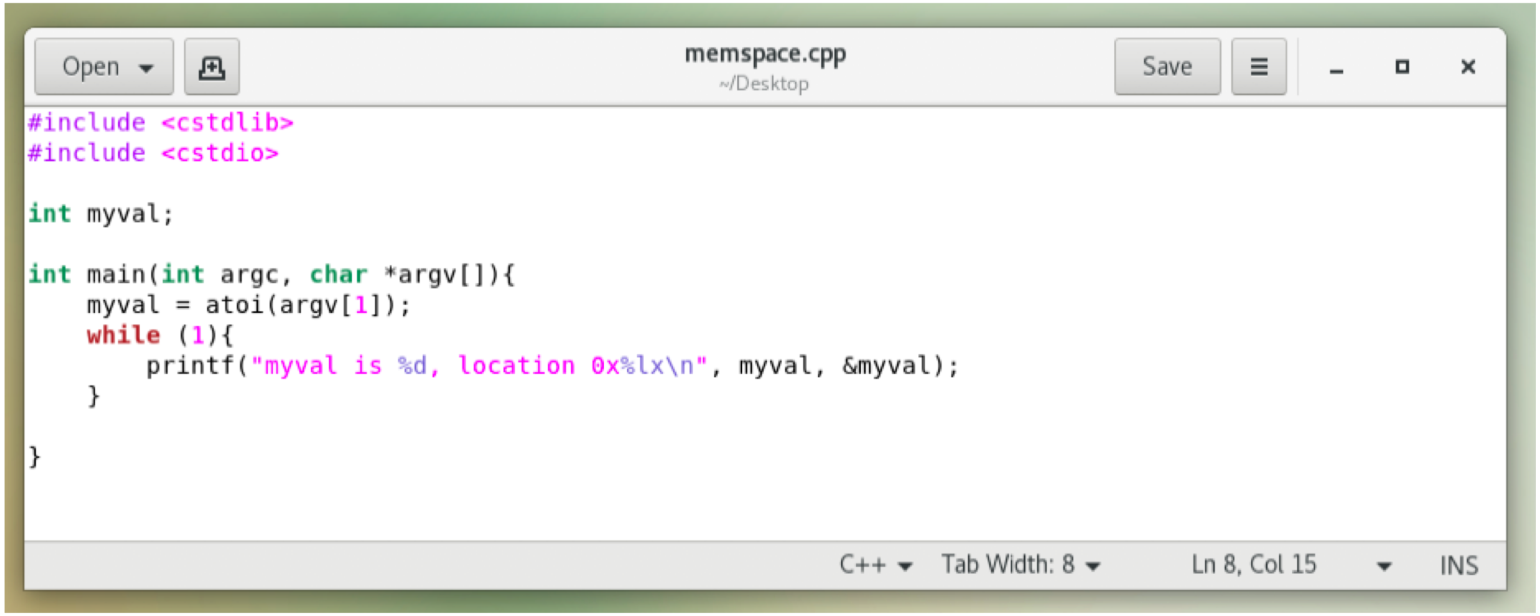
Memory addresses are respective.

Each process has such a memory space.



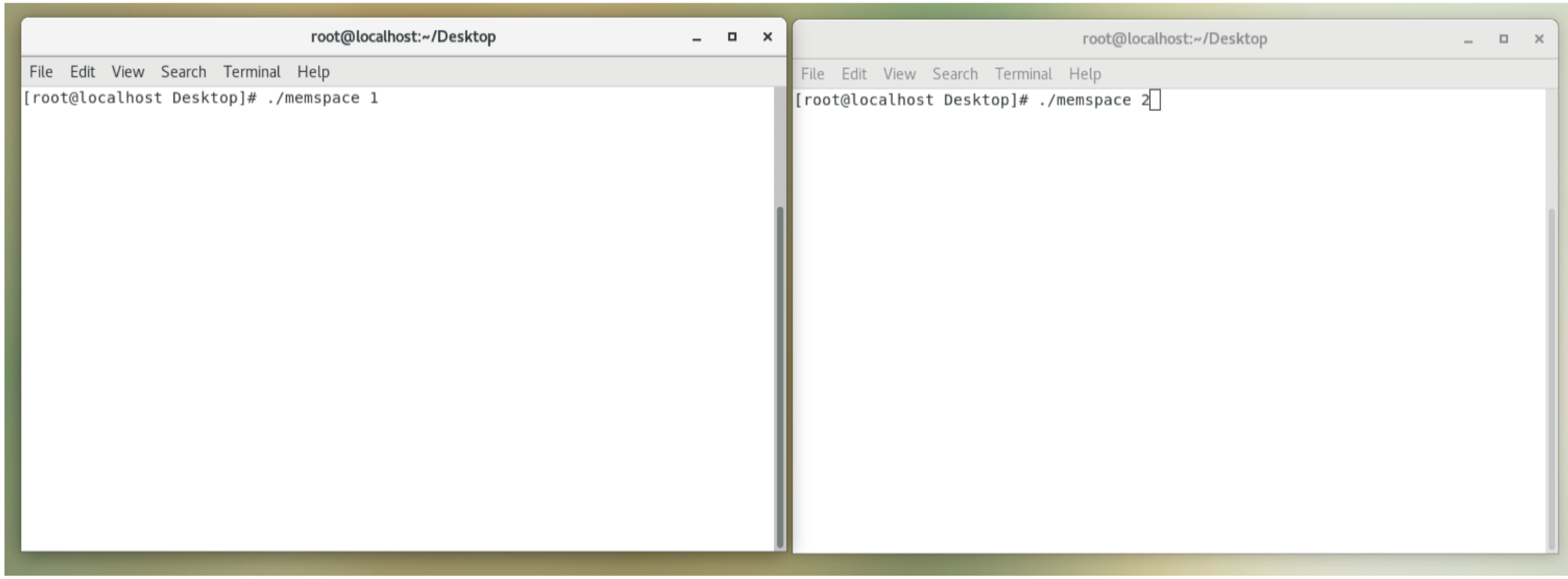
Process Address Space

Memory addresses are respective, not absolute.

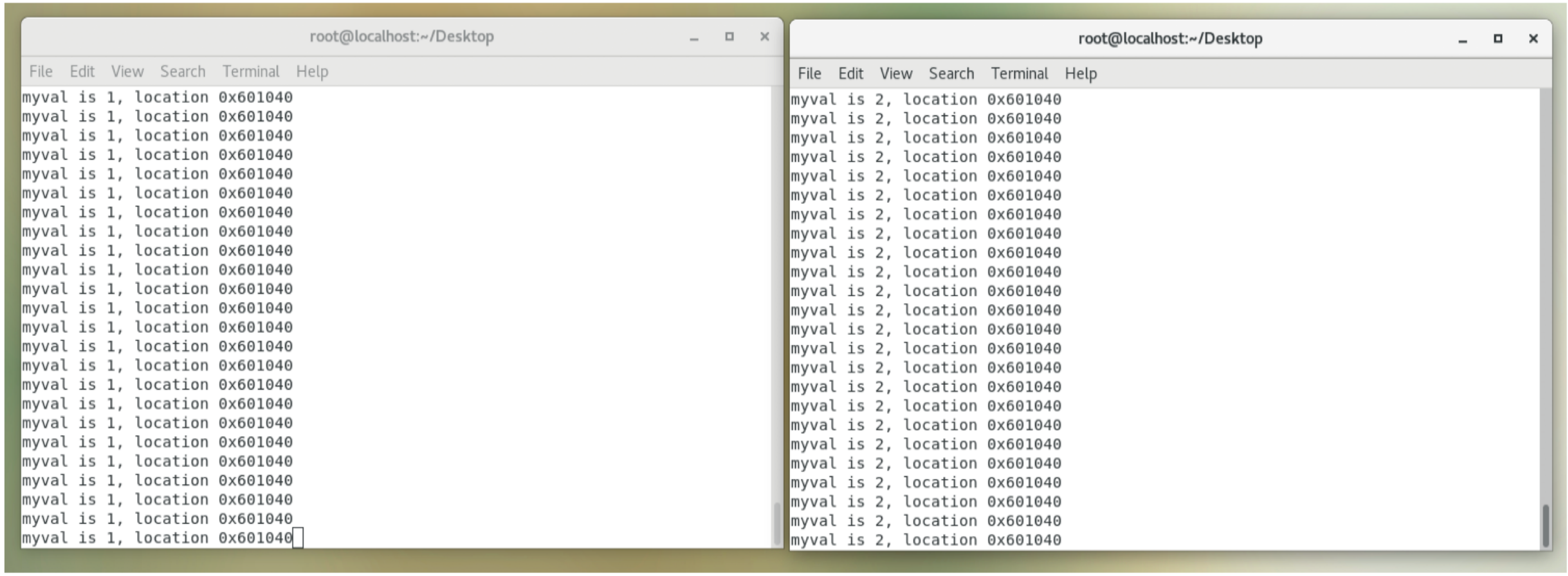


Acknowledgement: Prof. Yuanyuan Zhou at UCSD

Process Address Space



Process Address Space

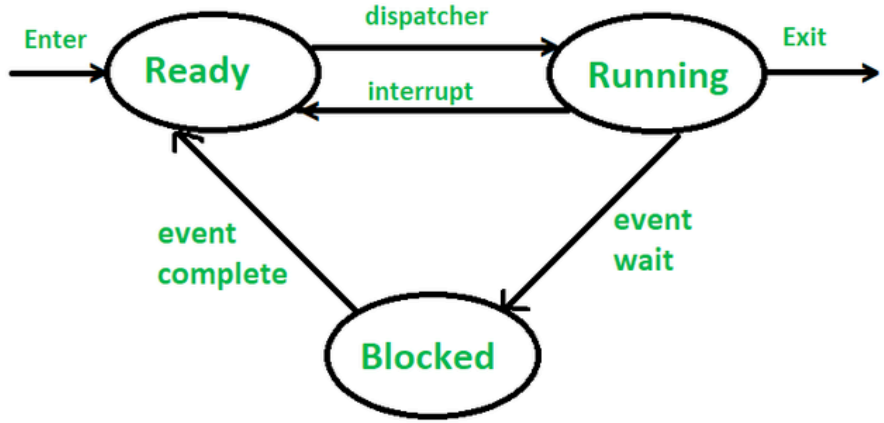


Their memory addresses are identical, however, the values are not the same.

Acknowledgement: Prof. Yuanyuan Zhou at UCSD

State of a process

- **Blocked:** waiting for something (for i/o result, message from another process)
- **Running:**
- **Ready:**



Context Switch

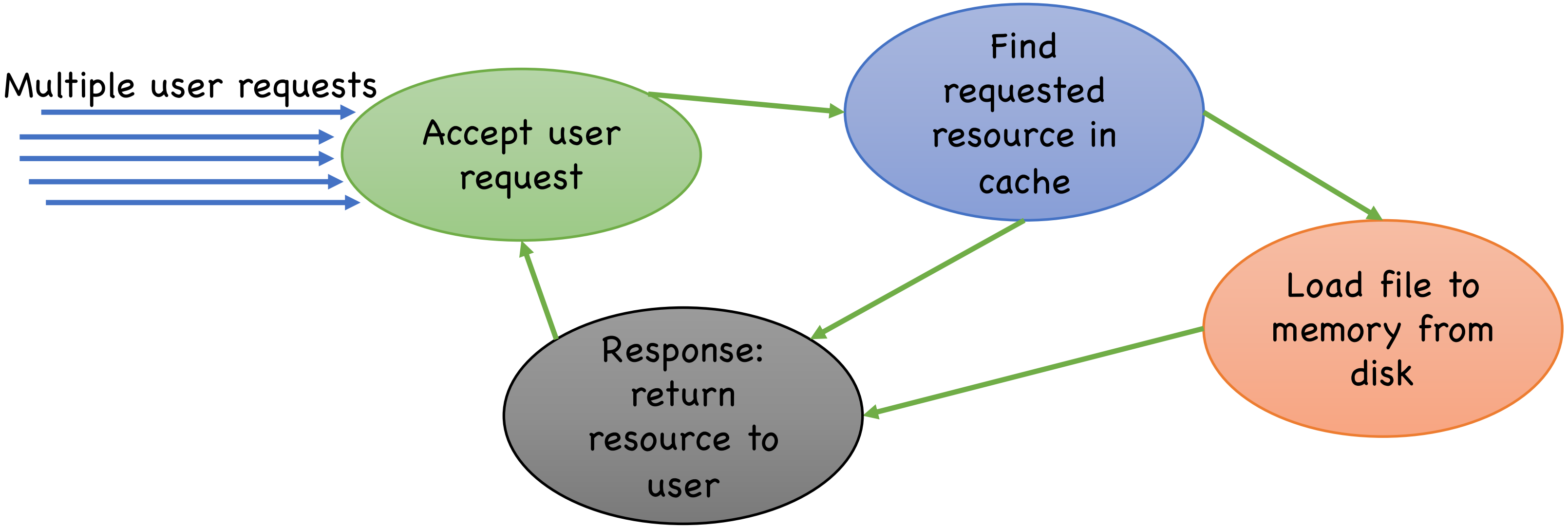
- We have only one CPU (one set of registers, one set of program counter, …)
- **What is happening:** When stopping a running process -> store the CPU context -> putthe CPU context of another ready process to CPU
- **Who is scheduling:** OS.

Type of a process

- **Computation intensive**
- **I/O intensive**

Introduction of thread

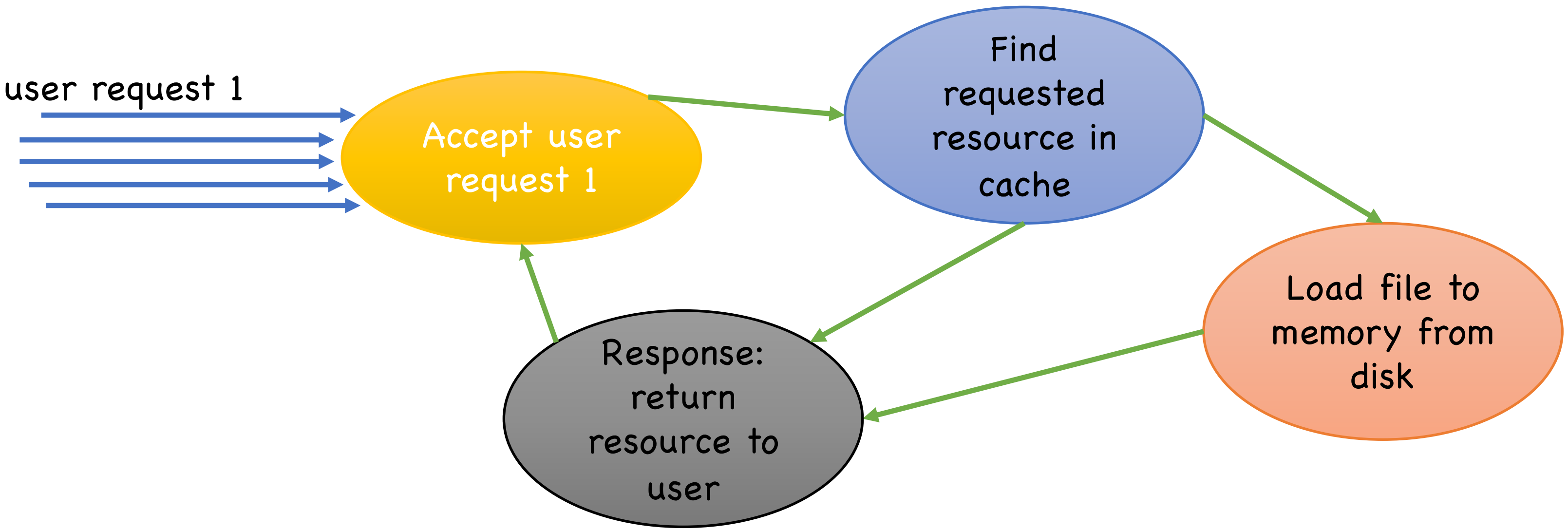
- **Necessity: Example of web server**
- **If we only have a single-thread process (a finite state machine).**



Acknowledgement: Prof. Xiangqun Chen (PKU)

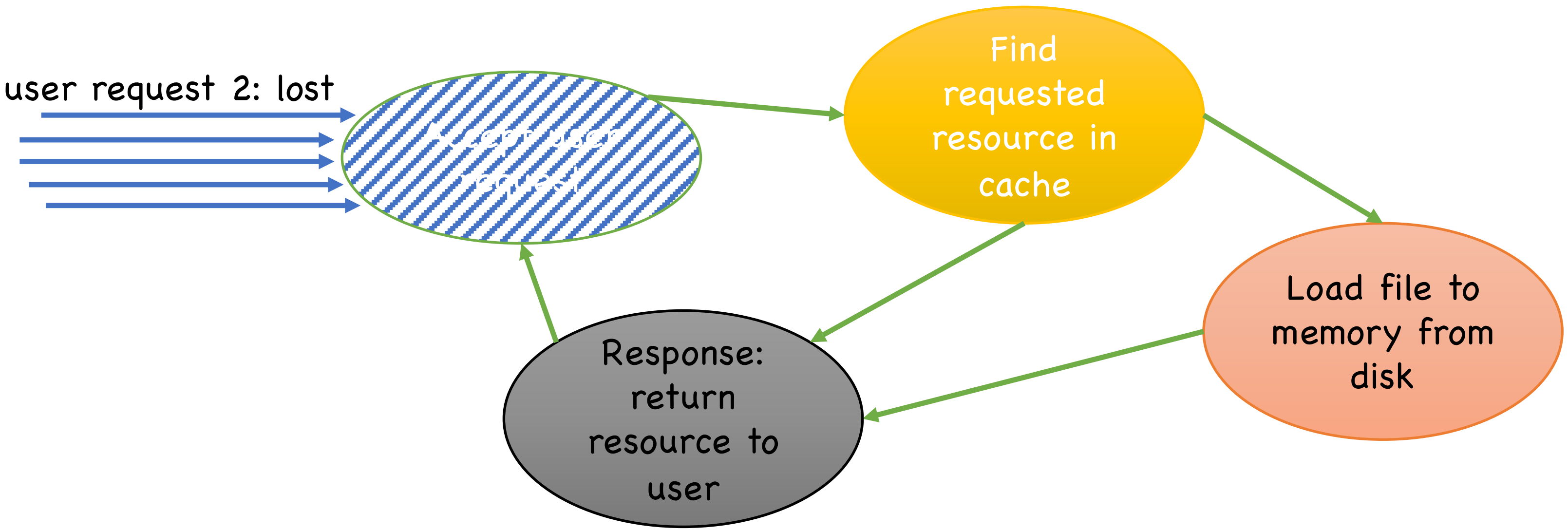
Introduction of thread

- **Necessity: Example of web server**
- **If we only have a single-thread process (a finite state machine).**



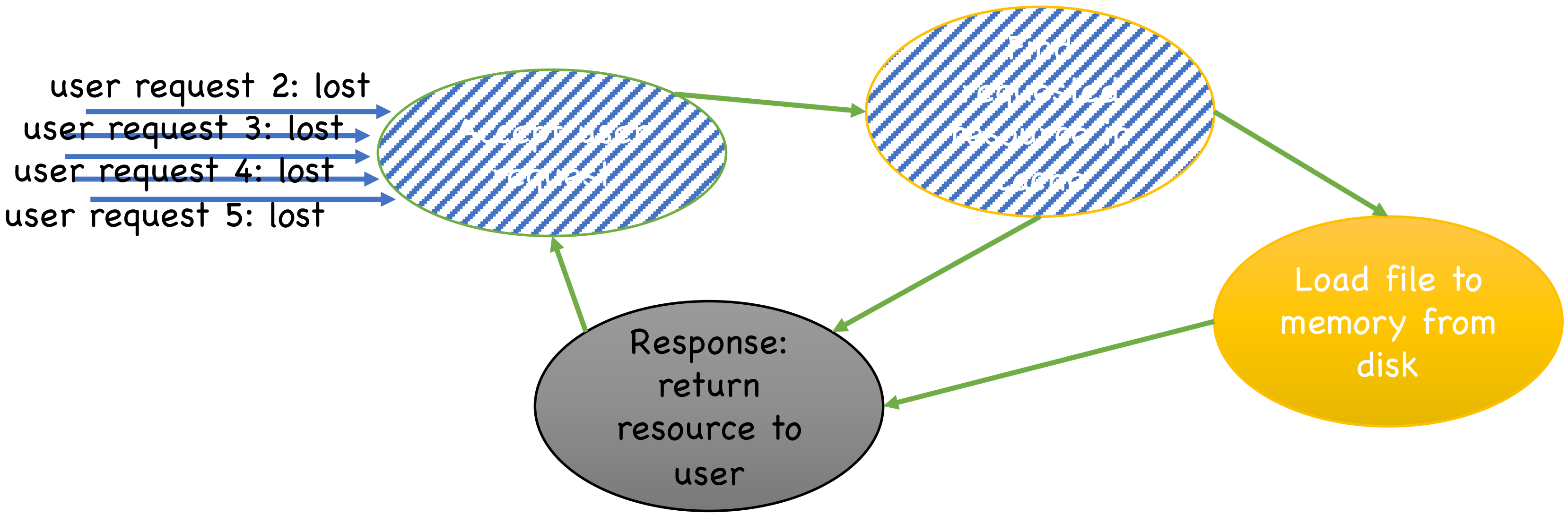
Introduction of thread

- **Necessity: Example of web server**
- **If we only have a single-thread process (a finite state machine).**



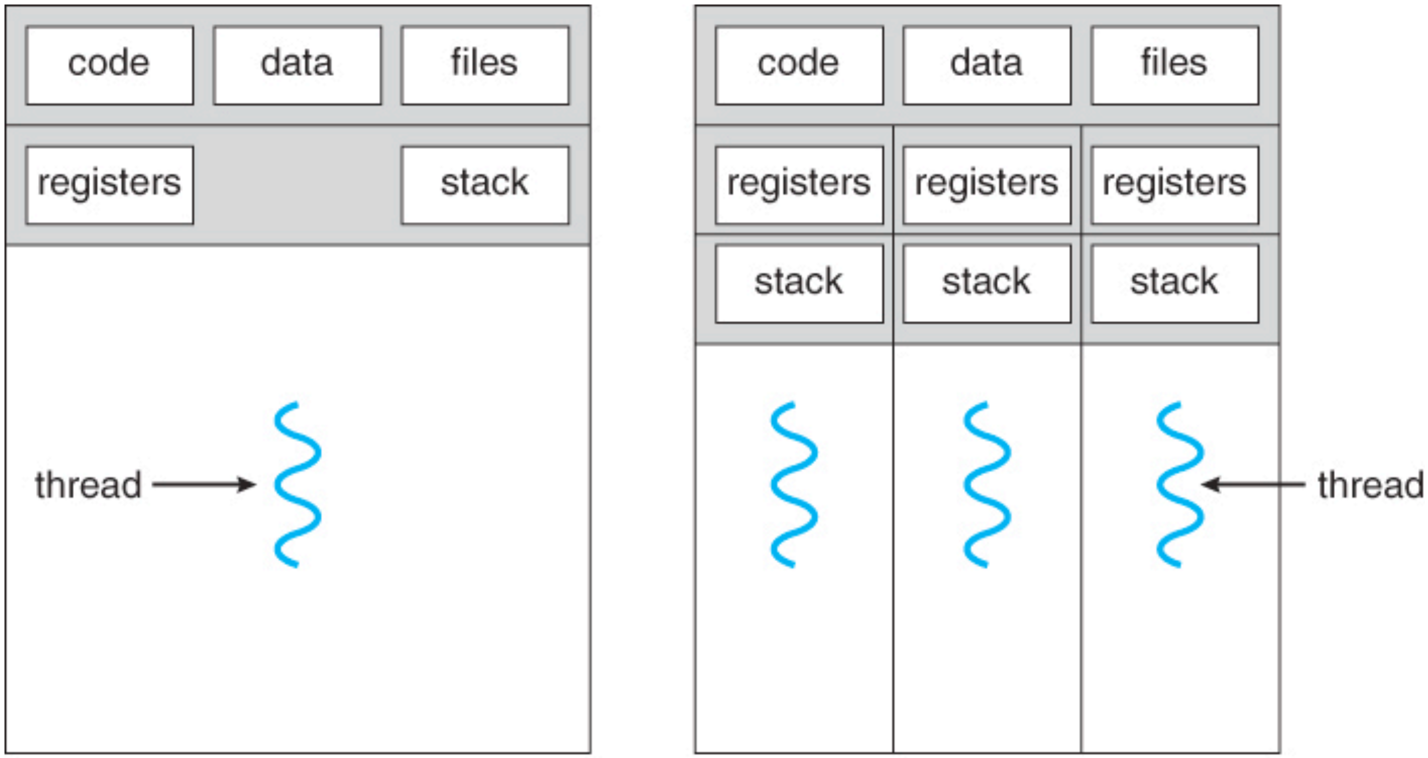
Introduction of thread

- **Necessity: Example of web server**
- **If we only have a single-thread process (a finite state machine).**



Possible solution: non-blocking finite state machine **Much more complex to design**

Thread



single-threaded process　　multithreaded process

Thread

A thread is a single sequence stream within a process.

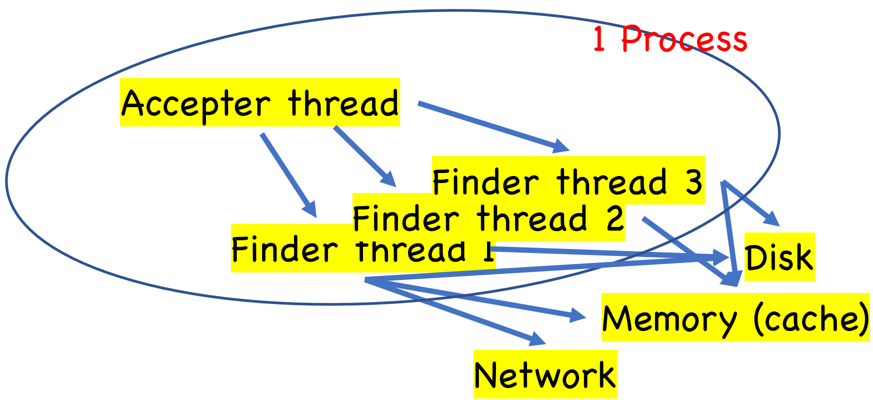
Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

- Threads are not independent from each other unlike processes. As a result, **threads shares with other threads their code section, data section and OS resources like open files and signals**.
- But, like processes, a thread has its own program counter (PC), a register set, and a stack space. (like CPU context)
- Threads are able to be scheduled by the operating system and run as independent entities within a process.
- **A process can have multiple threads**, all of which share the resources within a process and all of which execute within the same address space.

Acknowledgement: Lele Li (TA of CSC3150)

Introduction of thread

- **Necessity: Example of web server**
- **Multithread model**



Timeline
Accepter thread
Accepter thread
Accepter thread
Finder thread 1
Accepter thread
Finder thread 2
Accepter thread
Finder thread 1
Accepter thread
Finder thread 2
Accepter thread
Finder thread 1
Accepter thread

Fast switch

Acknowledgement: Prof. Xiangqun Chen (PKU)

Statistics

For example, the following table compares timing results for the **fork()** subroutine and the **pthreads_create()** subroutine. Timings reflect **50,000 process/thread creations**, were performed with the time utility, and units are in seconds, no optimization flags.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 375 MHz POWER3	61.94	3.49	53.74	7.46	2.76	6.79
IBM 1.5 GHz POWER4	44.08	2.21	40.27	1.49	0.97	0.97
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

http://www.cs.unibo.it/~ghini/didattica/sistop/pthreads_tutorial/POSIX_Threads_Programming.htm

Multithreading

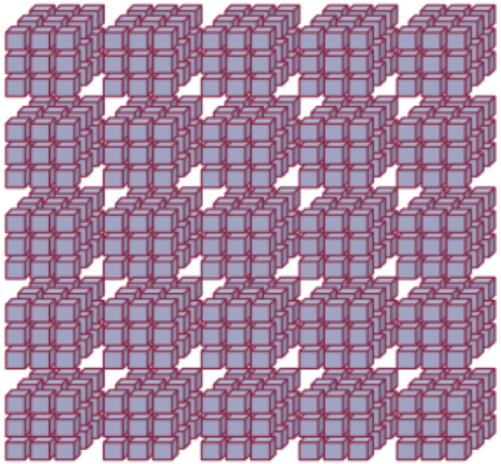
Necessity:

- Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads.
- Threads operate faster than processes due to following reasons:
- 1) Thread creation is much faster.
- 2) Context switching between threads is much faster.
- 3) Threads can be terminated easily
- 4) Communication between threads is faster.

Acknowledgement: Lele Li (TA of CSC3150)

Discussion

- Is multithreading always faster than multiprocessing?
  - **Computational intensive / I/O intensive**
- How about using **enormous amount of** degenerated CPU cores, which can only execute simple tasks --- **GPU**.



Up to 64*32=2048 CUDA threads can be active on the SM at a time.

GTX1060 GPU has 1280 sm cores.

Implement in C/C++

- multithreading is not supported by the C language standard.
- POSIX Threads (or Pthreads) is a POSIX standard for threads.
- Implementation of pthread is available with gcc/g++ compiler.

For source code: include pthread header file

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

For compilation: link pthread library

```
g++ -o pthread_hello pthread_hello.cpp -lpthread
```

For run time:

```
./pthread_hello
```

Implement in C/C++

- For most programming languages (java, c, c++), **threads** of a single process can run on **multiple cpu cores**.

Pthread API: Creating Threads

int pthread_create(pthread_t * thread, const pthread_attr_t * attr, void * (*start_routine)(void *), void *arg);

thread - returns the thread id. **(it is like an identity for each thread)**

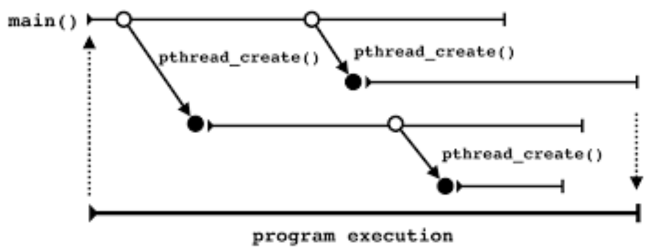
attr - Set to NULL if default thread attributes are used.

Attributes include:

- detached state (joinable? Default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED)
- scheduling policy (real-time? PTHREAD_INHERIT_SCHED,PTHREAD_EXPLICIT_SCHED,SCHED_OTHER)
- scheduling parameter
- inheritsched attribute (Default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED)
- (more attributes, over the scope of this course)

void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.

***arg** - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.



All threads are created by main thread.
Sub threads can create sub threads.

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

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Arguments explained

Create **identity** for each thread

Arguments passed in to the function should be a **pointer** to **void**. But now it is a char*

Convert to **void***

Convert to **pthread_t***

Function variable should be a **pointer** to **void**.

Pthread API: Passing multiple arguments

- Use struct to pass multiple arguments.

```
#include <pthread.h>
#include <cstdio>
#include <cstdlib>
#define NUM_THREADS 5

char *messages[NUM_THREADS];

struct thread_data {int thread_id; int  sum; char *message;};
struct thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void *threadarg) {
    int taskid, sum;
    char *hello_msg;
    struct thread_data *my_data;
    my_data = (struct thread_data *) threadarg;
    taskid = my_data->thread_id;
    sum = my_data->sum;
    hello_msg = my_data->message;
    printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
    pthread_exit(NULL);
}
```

Retrieve arguments

Load arguments

```
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int *taskids[NUM_THREADS];
    int rc, t, sum;
    sum = 0;
    messages[0] = "English: Hello World!";
    messages[1] = "French: Bonjour, le monde!";
    messages[2] = "Spanish: Hola al mundo";
    messages[3] = "Klingon: Nuq neH!";
    messages[4] = "German: Guten Tag, Welt!";
    for(t=0;t<NUM_THREADS;t++) {
        sum = sum + t;
        thread_data_array[t].thread_id = t;
        thread_data_array[t].sum = sum;
        thread_data_array[t].message = messages[t];
        printf("Creating thread %d\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
            &thread_data_array[t]);
        if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }
    pthread_exit(NULL);
}
```

Pthread API: Terminating Threads

void pthread_exit(void *retval);

This routine is used to **explicitly** exit a thread. Typically, the pthread_exit() routine is called after a thread has completed its work.

This method accepts a mandatory parameter **retval** which is the pointer to an integer that stores the return status of the thread terminated. The scope of this variable must be **global** so that any thread waiting to join this thread may read the return status.

If main() finishes before the threads it has created, and exits with pthread_exit(), the other threads will **continue** to execute. **Otherwise, they will be automatically terminated when main() finishes.**

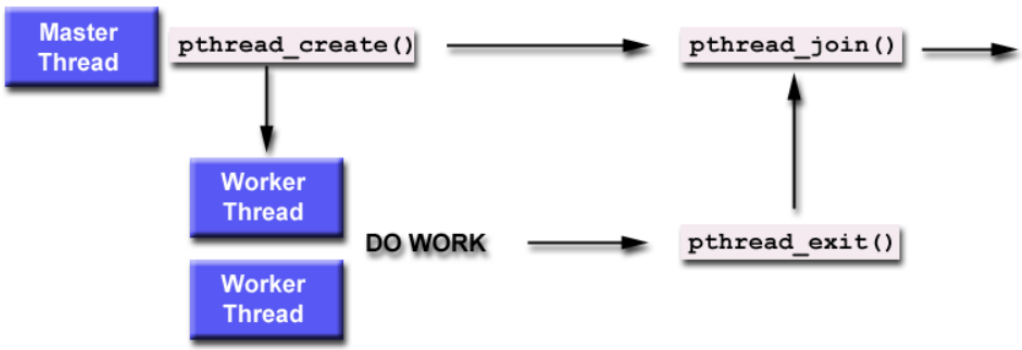
Recommendation: Use pthread_exit() to exit from all threads...especially main().

Acknowledgement: Lele Li (TA of CSC3150)

Pthread API: Synchronization

- **int pthread_join(pthread_t thread, void *retval);**

- **Joining** is one way to accomplish ***synchronization*** between threads.
- The pthread_join() subroutine **blocks the calling thread** until the **specified thread terminates**.



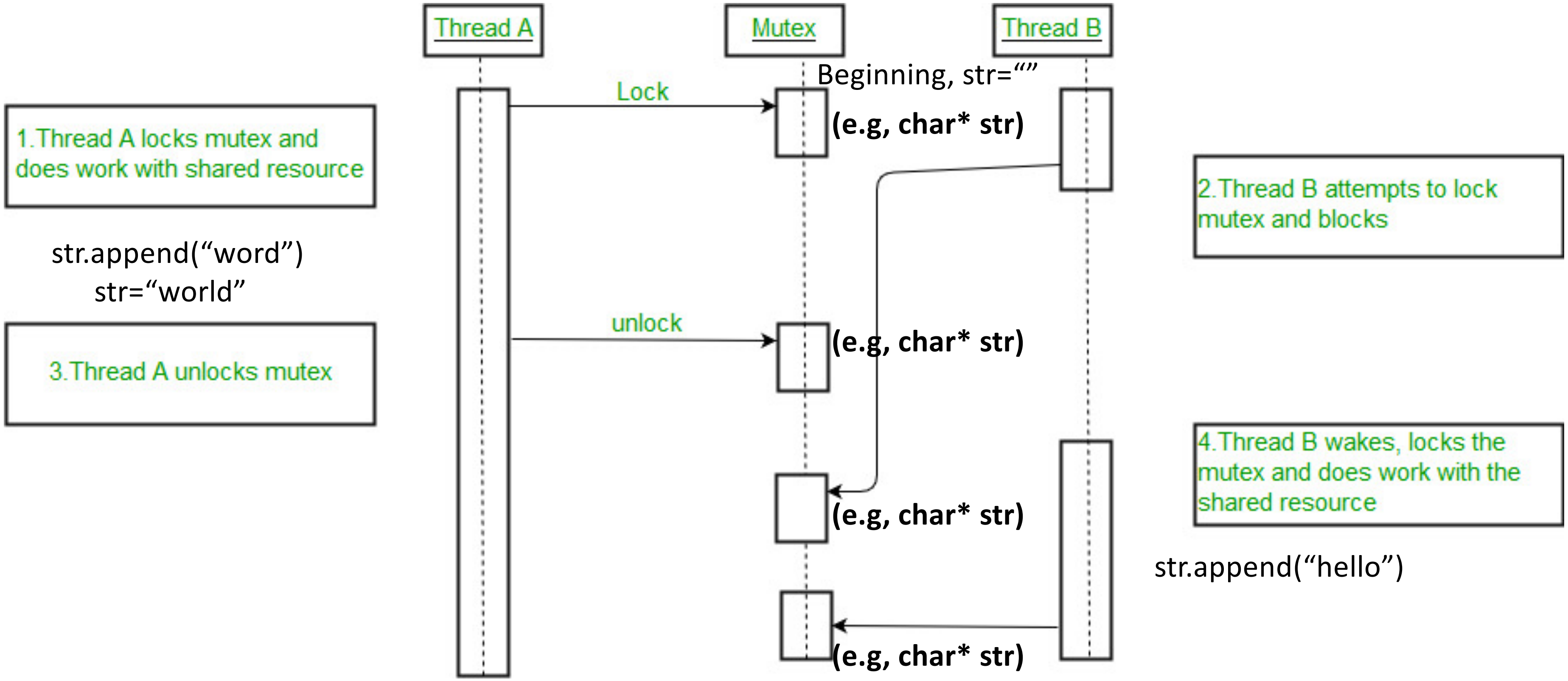
Pthread API: Mut(ual) ex(clusive) Lock

Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.

A mutex variable acts like a "lock" protecting access to a shared data resource.

Acknowledgement: Lele Li (TA of CSC3150)

A mutex variable acts like a "lock" protecting access to a shared data resource.



Finally, str=“world hello”

Acknowledgement: Lele Li (TA of CSC3150)

Pthread API: Mutex Lock

- **int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);**
- **int pthread_mutex_destroy(pthread_mutex_t *);**
- **int pthread_mutex_lock(pthread_mutex_t *mutex);** (block until unlock)
- **int pthread_mutex_trylock(pthread_mutex_t *mutex);** (non-blocking)
- **int pthread_mutex_unlock(pthread_mutex_t *mutex);**

It initialises the mutex referenced by *mutex* with attributes specified by *attr*.
If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object.
Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

Acknowledgement: Lele Li (TA of CSC3150)

Pthread API: Mutex Lock

pthread_mutex_lock() routine is used by a thread to **acquire a lock** on the specified mutex variable. **If the mutex is already locked by another thread, this call will block the calling thread until the mutex is unlocked**.

pthread_mutex_trylock() will **attempt to lock a mutex. However, if the mutex is already locked, the routine will return immediately with a "busy" error code**. This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

pthread_mutex_unlock() will **unlock a mutex if called by the owning thread**. Calling this routine is required after a thread has completed its use of protected data if other threads are to acquire the mutex for their work with the protected data. An **error** will be returned if:

- **If the mutex was already unlocked**
- **If the mutex is owned by another thread**

Acknowledgement: Lele Li (TA of CSC3150)

Pthread API: Signals & Condition Variable

Condition variables must be initialized before it is used:

- **int pthread_cond_init(pthread_cond_t *, const pthread_condattr_t *);**

Condition variables should be freed if it is no longer used:

- **int pthread_cond_destroy(pthread_cond_t *);**

Usage:

- **int pthread_cond_wait(pthread_cond_t *, pthread_mutex_t *); (block calling thread until signal received)**
- **int pthread_cond_signal(pthread_cond_t *); (send a signal to condition variable)**
- **int pthread_cond_broadcast(pthread_cond_t *);**

Acknowledgement: Lele Li (TA of CSC3150)

Pthread API: Signals & Condition Variable

pthread_cond_wait() blocks the calling thread until **the specified condition** is signalled. This routine should be called **while mutex is locked**, and it will **automatically release the mutex while it waits**.

pthread_cond_signal() routine is used **to signal (or wake up) another thread which is waiting** on the condition variable. It should be called after mutex is locked, and must unlock mutex in order for pthread_cond_wait() routine to complete.

pthread_cond_broadcast() routine should be used instead of pthread_cond_signal() if more than one thread is in a blocking wait state.

Acknowledgement: Lele Li (TA of CSC3150)

Project 2

- The TODO is to smartly allocate jobs to all the workers (process or thread).
- We have provided a template on https://github.com/bokesyo/CSC4005_2022Fall_Demo/tree/main/project2_template
- Computation function and GUI are ready for you, you only need to fill TODOs with your own implementation.
- Make comparison.

Project 2: Template Usage

- Don't worry about the mathematics part. We have prepared a completed atom function for computing the color given a point! Your only job in this project is to smartly partition all data to all workers.

```c
void compute(Point* p) {
    /*
    Give a Point p, compute its color.
    Mandelbrot Set Computation.
    It is not necessary to modify this function, because it is a completed
one.
    *** However, to further improve the performance, you may change this
function to do batch computation.
    */

    Compl z, c;
    float lengthsq, temp;
    int k;

    /* scale [0, X_RESN] x [0, Y_RESN] to [-1, 1] x [-1, 1] */
    c.real = ((float) p->x - X_RESN / 2) / (X_RESN / 2);
    c.imag = ((float) p->y - Y_RESN / 2) / (Y_RESN / 2);

    /* the following block is about math. */

    z.real = z.imag = 0.0;
    k = 0;

    do {
        temp = z.real*z.real - z.imag*z.imag + c.real;
        z.imag = 2.0*z.real*z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real*z.real+z.imag*z.imag;
        k++;
    } while (lengthsq < 4.0 && k < max_iteration);

    /* math block end */

    p->color = (float) k / max_iteration;

}
```

A atom function to compute the color of a specific point.

It is not necessary to modify this function, because it is a completed one.
However, to further improve the performance, you may change this function to do batch computation.

In this template, we have some `#ifdef GUI` in `asg2.h`.

```
#ifdef GUI
#include <GL/glut.h>
#include <GL/gl.h>
#include <GL/glu.h>
#endif

...

#ifdef GUI
void plot() {
...
}
#endif
```

we also have some `#ifdef GUI` in source code.

```
#ifdef GUI
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(500, 500);
glutInitWindowPosition(0, 0);
glutCreateWindow("Sequential");
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glMatrixMode(GL_PROJECTION);
gluOrtho2D(0, X_RESN, 0, Y_RESN);
glutDisplayFunc(plot);
#endif

...

#ifdef GUI
glutMainLoop();
#endif
```

Macro usage in header file and source code

To control compilation result.

Run your job on HPC cluster: Batch mode

mpi **sbatch** script

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                       # Run all processes on a single node
#SBATCH --ntasks=20                     # number of processes = 20
#SBATCH --cpus-per-task=1      # Number of CPU cores allocated to each process
(please use 1 here, in comparison with pthread)
#SBATCH --partition=Project            # Partition name: Project or Debug
(Debug is default)

cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project2_template/
mpirun -np 4 ./mpi 1000 1000 100
mpirun -np 20 ./mpi 1000 1000 100
mpirun -np 40 ./mpi 1000 1000 100
```

Run your job on HPC cluster: Batch mode

pthread **sbatch** script

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                    # Run all processes on a single node
#SBATCH --ntasks=1                   # number of processes = 1
#SBATCH --cpus-per-task=20      # Number of CPU cores allocated to each
process
#SBATCH --partition=Project            # Partition name: Project or Debug
(Debug is default)

cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project2_template/
./pthread 1000 1000 100 4
./pthread 1000 1000 100 20
./pthread 1000 1000 100 40
./pthread 1000 1000 100 80
./pthread 1000 1000 100 120
./pthread 1000 1000 100 200
...
```

Run your job on HPC cluster: Batch mode

pthread **sbatch** script

```
#!/bin/bash
#SBATCH --job-name=your_job_name # Job name
#SBATCH --nodes=1                          # Run all processes on a single node
#SBATCH --ntasks=1                         # number of processes = 1
#SBATCH --cpus-per-task=20      # Number of CPU cores allocated to each
process
#SBATCH --partition=Project            # Partition name: Project or Debug
(Debug is default)

cd /nfsmnt/119010355/CSC4005_2022Fall_Demo/project2_template/
./pthread 1000 1000 100 4
./pthread 1000 1000 100 20
./pthread 1000 1000 100 40
./pthread 1000 1000 100 80
./pthread 1000 1000 100 120
./pthread 1000 1000 100 200
...
```

Run your job on HPC cluster: Batch mode

Finally:

To submit your job, use

```
sbatch xxx.sh
```

Interactive mode

Interactive: salloc

If you want to run your program using interactive mode, use

For MPI porgram, we have learned before:

```
salloc -n20 -c1 # -c1 can be omitted.
mpirun -np 20 ./mpi 1000 1000 100
```

For pthread program,

```
salloc -n1 -c20 -p Project # we have only 1 process, 20 is the number of cores allocated per process.
srun ./pthread 1000 1000 100 20 # 20 is the number of threads.
```

Thank you!