Multithreading

Saptarshi Ghosh and Mainack Mondal

CS39002

Spring 2020-21

Topics for this lecture

- What is a thread?
- Why do you need threads?
- How are threads used in real-world?
- Multithreading models
- POSIX Pthread library

Topics for this lecture

- What is a thread?
- Why do you need threads?
- How are threads used in real-world?
- Multithreading models
- POSIX Pthread library

What is a thread?

- Process is a program in execution with a single *thread* of control

What is a thread?

- Process is a program in execution with a single *thread* of control
- All modern OS allows a process to have multiple threads of control

What is a thread?

- Process is a program in execution with a single *thread* of control
- All modern OS allows a process to have multiple threads of control

- Multiple tasks within an application can be implemented by separate threads, e.g., in a word processor
    - Update display
    - Respond to keystrokes from user
    - Spell checking

How is a thread created?

- Can be considered a basic unit of CPU utilization
  - Unique thread ID, Program counter (PC), register set & stack

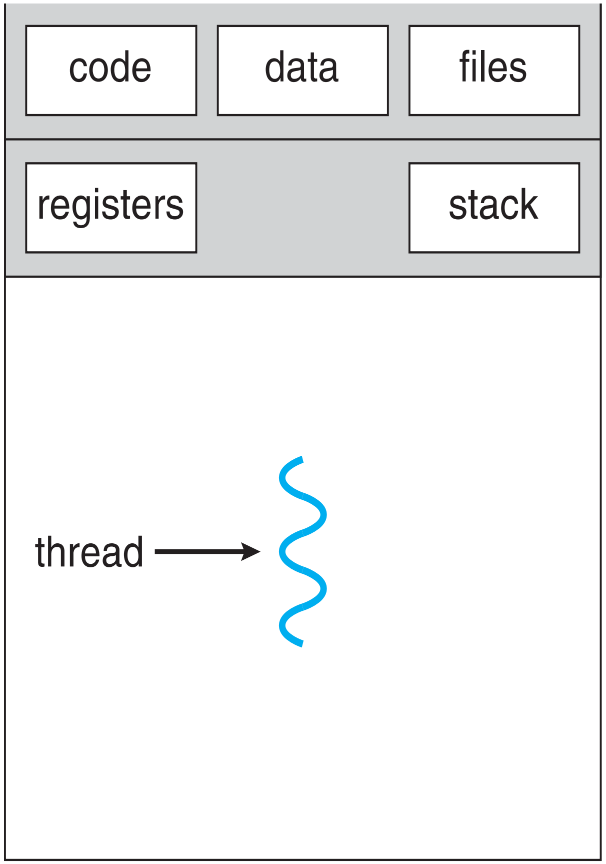
How is a thread created?

- Can be considered a basic unit of CPU utilization
  - Unique thread ID, Program counter (PC), register set & stack
  - Shares with other threads from same process the code section, data section and other OS resources like open files

How is a thread created?

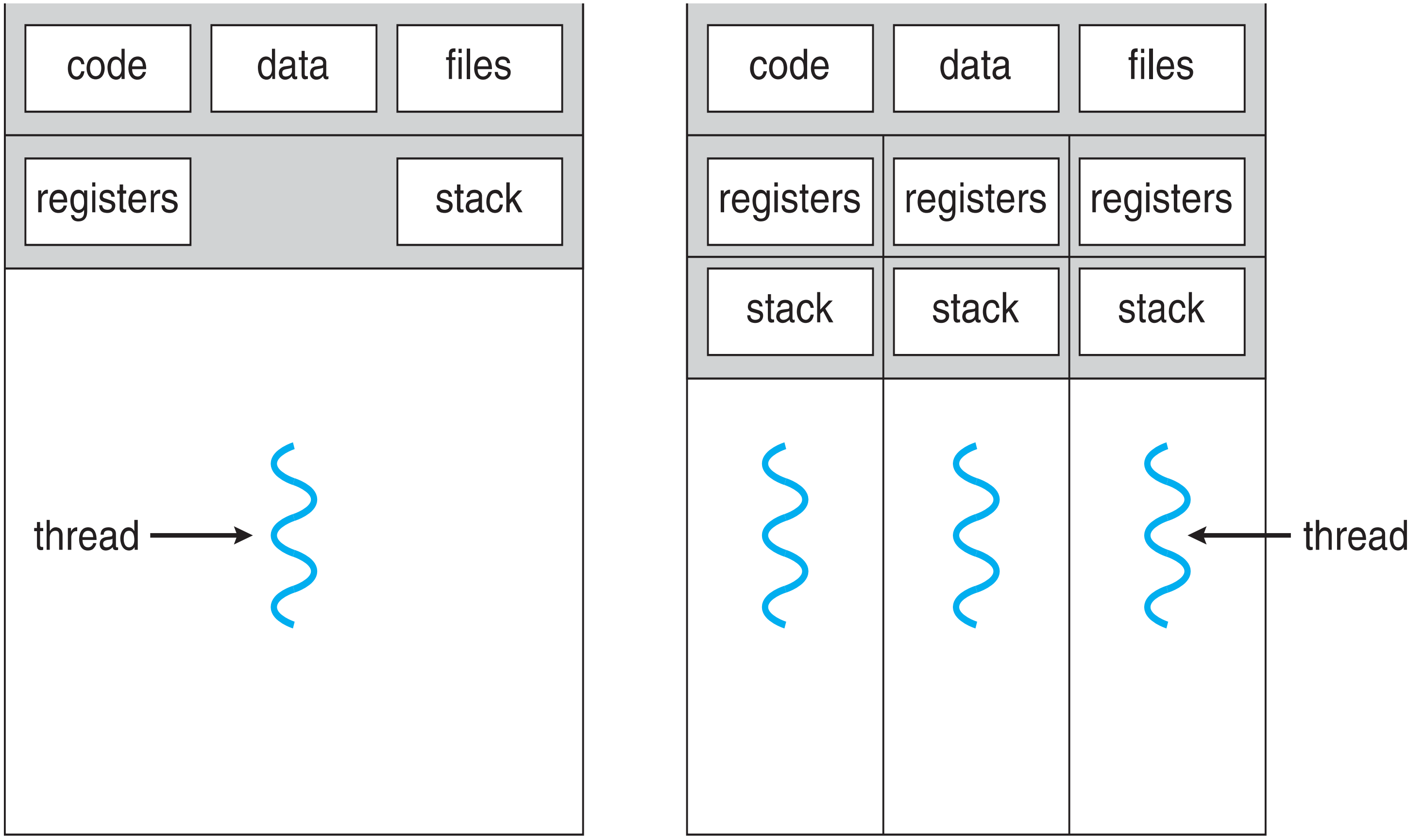
- Can be considered a basic unit of CPU utilization
  - Unique thread ID, Program counter (PC), register set & stack
  - Shares with other threads from same process the code section, data section and other OS resources like open files
  - Essentially all threads of the same process share the same virtual memory address space

Comparison: single and multi threaded processes



single-threaded process

Comparison: single and multi threaded processes



single-threaded process

multithreaded process

Topics for this lecture

- What is a thread?
- Why do you need threads?
- How are threads used in real-world?
- Multithreading models
- POSIX Pthread library

Thread: The benefits

- Context switching among threads of same process is faster
  - OS needs to reset/store less memory locations/registers

Thread: The benefits

- Context switching among threads of same process is faster
  - OS needs to reset/store less memory locations/registers
- Responsiveness is better (important for interactive applications)
  - E.g., even if part of process is busy the interface still works

Thread: The benefits

- Context switching among threads of same process is faster
  - OS needs to reset/store less memory locations/registers
- Responsiveness is better (important for interactive applications)
  - E.g., even if part of process is busy the interface still works
- Resource sharing is better for *peer* threads
  - Many possible threads of activity in same address space
  - Sharing variable is more efficient than pipe, shared memory

Thread: The benefits

- Context switching among threads of same process is faster
  - OS needs to reset/store less memory locations/registers
- Responsiveness is better (important for interactive applications)
  - E.g., even if part of process is busy the interface still works
- Resource sharing is better for *peer* threads
  - Many possible threads of activity in same address space
  - Sharing variable is more efficient than pipe, shared memory
- Thread creation: Process creation is **heavy-weight** while thread creation is **light-weight** (10-30 times faster than process creation)

Thread: The benefits

- Context switching among threads of same process is faster
  - OS needs to reset/store less memory locations/registers
- Responsiveness is better (important for interactive applications)
  - E.g., even if part of process is busy the interface still works
- Resource sharing is better for *peer* threads
  - Many possible threads of activity in same address space
  - Sharing variable is more efficient than pipe, shared memory
- Thread creation: Process creation is **heavy-weight** while thread creation is **light-weight** (10-30 times faster than process creation)
- Better scalability for multiprocessor / multicore architecture

Topics for this lecture

- What is a thread?
- Why do you need threads?
- How are threads used in real-world?
- Multithreading models
- POSIX Pthread library

Thread: The applications

- A typical application is implemented as a separate process with multiple threads of control
  - Ex 1: A web browser
  - Ex 2: A web server
  - Ex 3: An OS

Thread example 1: Web browser

- Think of a web browser (e.g., chrome)
  - Thread 1: retrieve data
  - Thread 2: display image or text (render)
  - Thread 3: waiting for user input (your password)
  - …

Thread example 2: Web server

- A single instance of web server (apache tomcat, nginx) may be required to perform several similar tasks
    - One thread accepts request over network (continues listening for new connection requests)
    - New threads service requests: one thread per request
    - The main process creates these threads

Thread example 3: OS

- Most OS kernels are multithreaded
  - Several threads operate in kernel
  - Each thread performing a specific task
  - E.g., managing memory, managing devices, handling interrupts etc.

Topics for this lecture

- What is a thread?
- Why do you need threads?
- How are threads used in real-world?
- Multithreading models
- POSIX Pthread library

User threads and kernel threads

- User threads: managed by user-level threads library
  - A few well-established primary thread libraries
  - POSIX Pthreads, Windows threads, Java threads

User threads and kernel threads

- User threads: managed by user-level threads library
  - A few well-established primary thread libraries
  - POSIX Pthreads, Windows threads, Java threads
- Kernel threads - supported and managed by the Kernel
  - Exists virtually in all general-purpose OS
  - Windows, Linux, Mac OS X

User threads and kernel threads

- User threads: managed by user-level threads library
  - A few well-established primary thread libraries
  - POSIX Pthreads, Windows threads, Java threads
- Kernel threads - supported and managed by the Kernel
  - Exists virtually in all general-purpose OS
  - Windows, Linux, Mac OS X

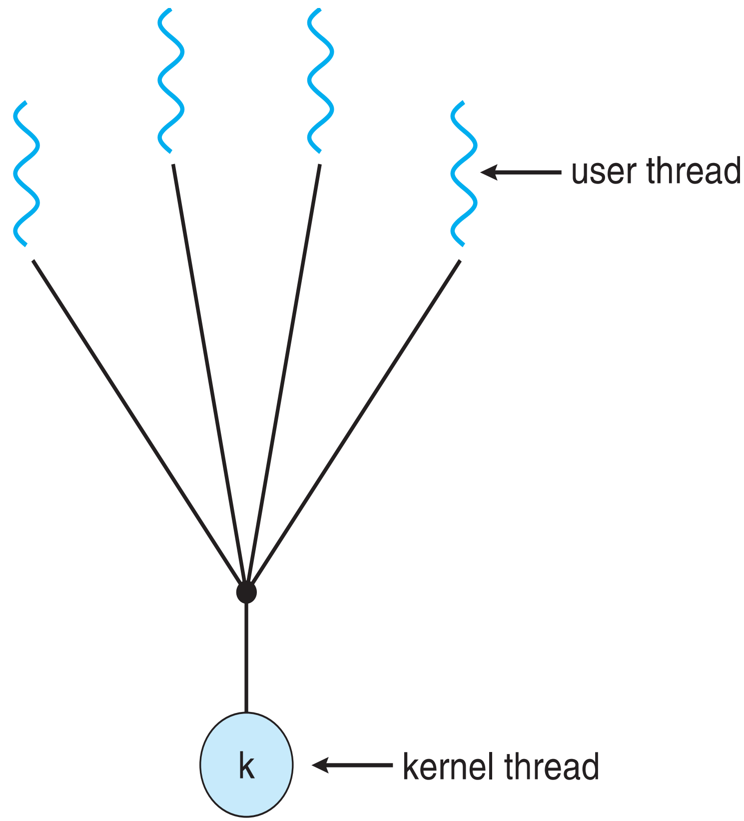
As you might have guessed: user threads will ultimately need kernel thread support

Multithreading Models

- There are multiple models to map user threads to kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

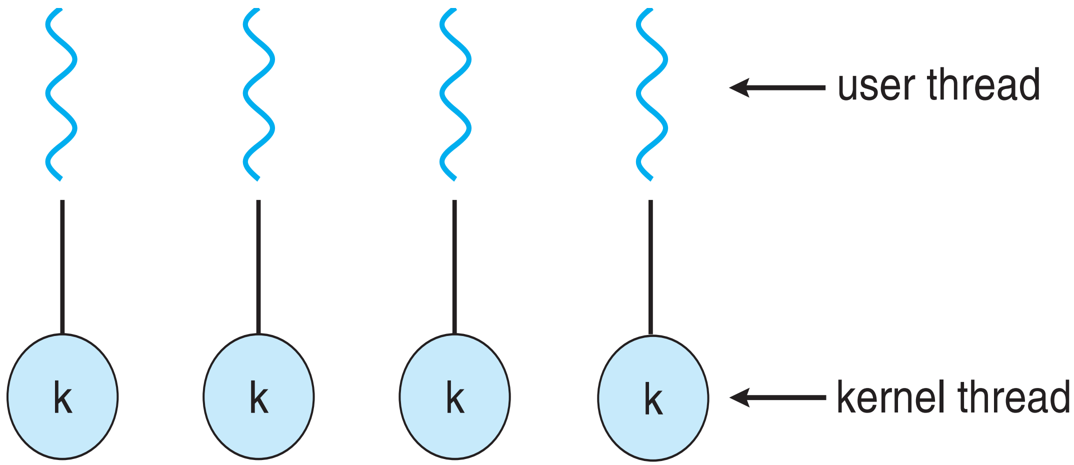
Many-to-One

- Many user-level threads mapped to single kernel thread
- User threads managed by thread library in user space
- Blocking call by one thread causes all threads in process to block
- Multiple threads may not run in parallel on multicore system because only one can access kernel at a time
- Old model: Only few systems currently use this model



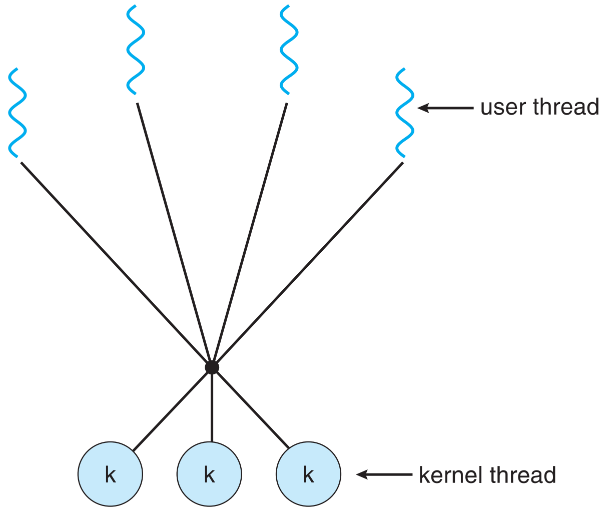
One-to-One

- Each user-level thread maps to one kernel thread
- A user-level thread creation -> a kernel thread creation
- More concurrency than many-to-one (even if one thread makes a blocking system call, others can run)
- #threads per process sometimes restricted due to overhead on kernel (of creating kernel threads)
- Windows, Linux



Many-to-Many Model

- Allows many user-level threads to be mapped to several (a smaller or equal number of) kernel threads
- Allows the OS to create a sufficient number of kernel threads
- If one thread performs a blocking system call, kernel can schedule another thread for execution
- Windows with the *ThreadFiber* package



Topics for this lecture

- What is a thread?
- Why do you need threads?
- How are threads used in real-world?
- Multithreading models
- POSIX Pthread library

Thread library

- Provides the programmer with an API for creating and managing threads
- Two ways of implementation
  - Provide the library in the user-space (all code and data structures of the library exist in user-space)
  - Provide a kernel-level library (code and data structures of the library exist in kernel-space; invoking a library function results in a system call)
- Some popular thread libraries: POSIX Pthreads, Windows, Java

Two strategies for using threads

- Asynchronous threading
  - Parent thread creates one or more child threads, and resumes execution
  - Parent and child threads execute concurrently (each thread runs independent of others)
  - Parent need not know when child threads terminate
  - Typically, very little data sharing between threads

Two strategies for using threads

- Asynchronous threading
  - Parent thread creates one or more child threads, and resumes execution
  - Parent and child threads execute concurrently (each thread runs independent of others)
  - Parent need not know when child threads terminate
  - Typically, very little data sharing between threads
- Synchronous threading
  - Parent thread creates one or more child threads, and then waits for all child threads to terminate before it resumes
  - Also called fork-join strategy
  - Typically involves significant data sharing among threads (e.g., parent combines results computed by child threads)

POSIX Pthreads: basics

- Actually a standard / specification (IEEE 1003.1c) that defines an API for thread creation and synchronization
  - POSIX: Portable Operating System Interface
  - Family of standards for maintaining OS compatibility
  - Basically, tells OS what function calls need to be supported
  - Increases portability
- May be provided either as user-level or kernel-level library
- All major thread libraries in unix are POSIX compatible

POSIX Pthreads: basics

- Include `pthread.h` in the main source file
- Compile program with `-lpthread`
  - `gcc -o test test.c -lpthread`
  - Otherwise, may not report compilation errors but calls will fail
- Global data: Any variable/data declared globally are shared among all threads of the same process
- Local data: Data local to a function stored in stack. Since each thread has own stack, each thread has own copy
- Threads begin execution in a specified function

POSIX Pthreads: an example

- A separate thread is created that calculates the sum of N natural numbers (N is an input)
- The parent thread waits for the child thread to end

The code

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

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){






}

void *runner (void *param){




}
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t tid;  // identifier for the thread that we will create
        pthread_attr_t attr;  // will store attributes of the thread (e.g., stack size)
        pthread_attr_init (&attr); // get default attributes



}

void *runner (void *param){





}
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init (&attr); // get default attributes
        pthread_create(&tid, &attr, runner, argv[1]); // create the thread


}

void *runner (void *param){




}
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init (&attr); // get default attributes
        pthread_create(&tid, &attr, runner, argv[1]); // create the thread
        pthread_join(tid, NULL); //wait for the thread to exit (fork-join strategy)

}

void *runner (void *param){




}
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init (&attr); // get default attributes
        pthread_create(&tid, &attr, runner, argv[1]); // create the thread
        pthread_join(tid, NULL); //wait for the thread to exit
        printf("\n sum = %d", sum); // print accumulated sum
}

void *runner (void *param){




}
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init (&attr); // get default attributes
        pthread_create(&tid, &attr, runner, argv[1]); // create the thread
        pthread_join(tid, NULL); //wait for the thread to exit
        printf("\n sum = %d", sum); // print accumulated sum
}

void *runner (void *param){
        int I , N = atoi(param); // get input value
        sum = 0;


}
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init (&attr); // get default attributes
        pthread_create(&tid, &attr, runner, argv[1]); // create the thread
        pthread_join(tid, NULL); //wait for the thread to exit
        printf(“\n sum = %d”, sum); // print accumulated sum
}

void *runner (void *param){
        int I , N = atoi(param); // get input value
        sum = 0;
        for(i = 1; i<=N;i++){sum = sum+i;}

}
```

The code

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

int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t tid;
        pthread_attr_t attr;
        pthread_attr_init (&attr); // get default attributes
        pthread_create(&tid, &attr, runner, argv[1]); // create the thread
        pthread_join(tid, NULL); //wait for the thread to exit
        printf("\n sum = %d", sum); // print accumulated sum
}

void *runner (void *param){
        int i , N = atoi(param); // get input value
        sum = 0;
        for(i = 1; i<=N;i++){sum = sum+i;}
        pthread_exit(0); // terminate the thread
}
```

You can also create many threads

```
#include<stdio.h>
#include<pthread.h>
#define N_THR 10
Int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t mythreads[N_THR];
        …
        …
        for (int i=0; i< N_THR; i++)
                pthread_create(&mythreads[i], &attr, runner, argv[1]); // create the
threads




}

void *runner (void *param){
…
}
```

You can also create many threads

```
#include<stdio.h>
#include<pthread.h>
#define N_THR 10
Int sum; // data shared over threads

void *runner (void *param); // child process calls this

int main(int argc, char *argv[]){
        pthread_t mythreads[N_THR];
        …
        …
        for (int i=0; i< N_THR; i++)
                pthread_create(&mythreads[i], &attr, runner, argv[1]); // create the threads
         for (int i=0; i< N_THR; i++)
                pthread_join(mythreads[i], NULL); //wait for the threads to exit
        printf("\n sum = %d", sum); // print accumulated sum
}

void *runner (void *param){
…
}
```

exit() Vs. pthread_exit()

- exit() kills all threads
  - Including the main() thread
  - pthread_exit() only kills the running thread but keep the task alive

Thread Attributes

- Type: `pthread_attr_t` (see previous example)
- Attributes define the state of the new thread
- State: system scope, joinable, stack size, inheritance
- Default behaviors with `NULL` in `pthread_create()`

  int **pthread_attr_init**(&attr);

  pthread_attr_{set/get}{attribute}

- Example:

```
pthread_attr_t attr;
pthread_attr_init(&attr);
```