Multithreading

Indranil Sen Gupta (odd section)
and Mainack Mondal (even section)
CS39002

Spring 2019-20

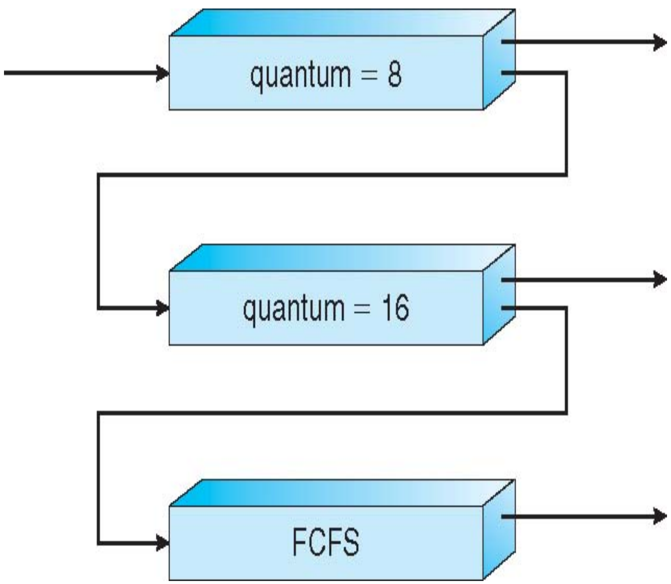
But first, we will do some problem solving and recap

Scheduling criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
- Burst time – amount of time a process is executed

Recap: Multi level feedback queue

- Three queues:
  - Q_0 – RR with time quantum (δ) 8 ms
  - Q_1 – RR with $\delta = 16$ms
  - Q_2 – FCFS



- A process in Q1 can execute only when Q0 is empty
- A process in Q0 can pre-empt a process in Q1 or Q2
- If the CPU burst of a process exceeds δ its moved to lower priority queue

Issue with Multi level feedback queue scheduling

- Long running processes may starve
  - Permanent demotion of priority hurts processes that change their behavior (e.g., lots of computation only at beginning)
  - Eventually all long-running processes move to FCFS

Issue with Multi level feedback queue scheduling

- Long running processes may starve
    - Permanent demotion of priority hurts processes that change their behavior (e.g., lots of computation only at beginning)
    - Eventually all long-running processes move to FCFS

- Solution
    - periodic priority boost: all processes moved to high priority queue
    - Priority boost with aging: recompute priority based on scheduling history of a process

Ex 1: First Come First Serve scheduling (FCFS)

Example 1

Process	P1	P2	P3
Arrival time	0	0	0
CPU burst	24ms	3ms	3ms

Draw Gantt chart and calculate average waiting time for two schedules: P1, P2, P3 and P2, P3, P1 (Ans: 17 ms and 3 ms)

Ex 2: Another FCFS

Example 2

Process	P1	P2	P3	P4	P5
Arrival time	0	2ms	3ms	5ms	9ms
CPU burst	3ms	3ms	2ms	5ms	3ms

Draw Gantt chart and calculate average waiting time

(Ans: 11/5 ms)

Ex 3: SJF

Process	P1	P2	P3	P4
Arrival time	0	0	0	0
CPU burst	6ms	8ms	7ms	3ms

What is the SJF schedule and corresponding wait time?

Compare with the following FCFS schedule: P1, P2, P3, P4

(Ans: SJF – 7 ms and FCFS – 10.25 ms)

Ex 4: Shortest remaining time first

- Pre-emptive version of SJF
  - A smaller CPU burst time process can evict a running process

Process	P1	P2	P3	P4
Arrival time	0	1ms	2ms	3ms
CPU burst	8ms	4ms	9ms	5ms

- Draw preemptive gantt chart and computing waiting time.

(Ans: 6.5 ms)

Ex 5: Priority scheduling

- A priority is assigned to each process
  - CPU is allotted to the process with highest priority
  - SJF is a type of priority scheduling

Process	P1	P2	P3	P4	P5
Arrival time	0	0	0	0	0
CPU burst	10ms	1ms	2ms	1ms	5ms
Priority	3	1	4	5	2

What is the average waiting time?

(Ans: 8.2 ms)

Ex 6: RR scheduling

Example:

Process	P1	P2	P3
Arrival time	0	0	0
CPU burst	24ms	3ms	3ms

If time quantum δ = 4 ms, then what is the avg. wait time?
(schedule P1, P2, P3,…)

(Ans: 5.66ms)

Try this Exercise

Process	P1	P2	P3	P4
Arrival time	0	0	0	0
CPU burst	6ms	3ms	1ms	7ms

Compute average turnaround time for δ = 1,2,3,4,5,6,7ms

Compute average wait time for δ = 1,2,3,4,5,6,7ms

Assume the schedule is P1, P2, P3, P4

Now let’s go into multithreading

Rest of today's class

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

Rest of today's class

- 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 single *thread* of control

What is a thread?

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

What is a thread?

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

- Multiple tasks within an application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request

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?

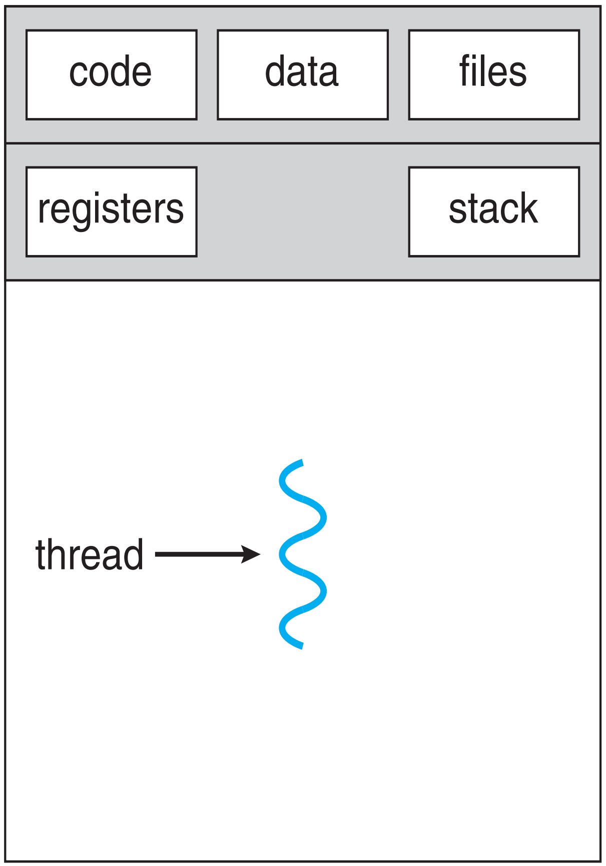
- Can be considered a basic unit of CPU utilization
    - Unique thread ID, Program counter (PC), register set & stack
    - Shares with other threads from same process the code section, data section and other OS resources like open files

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 same virtual memory address space

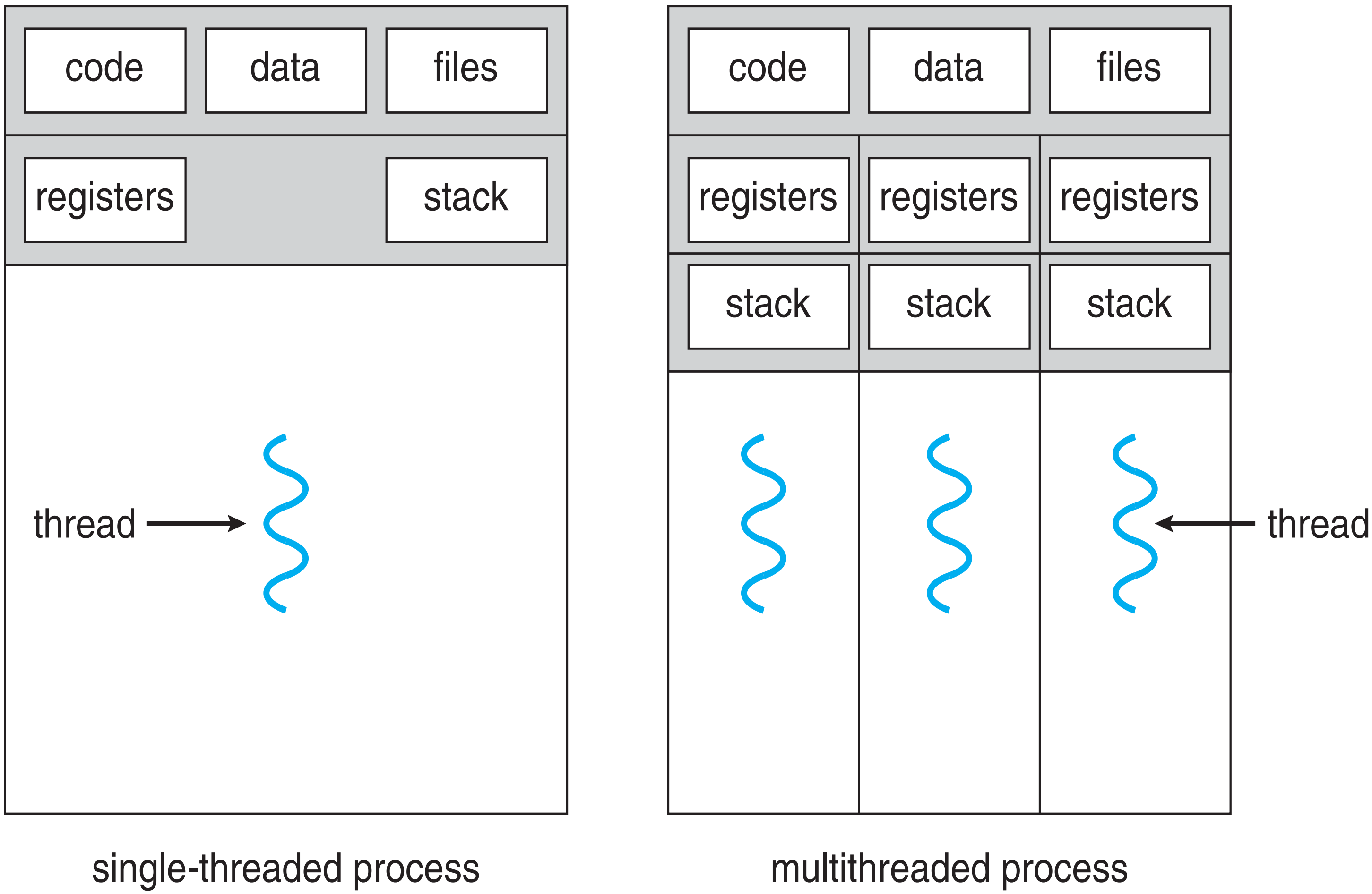
- Process creation is heavy-weight while thread creation is light-weight

Comparison: single and multi threaded processes



single-threaded process

Comparison: single and multi threaded processes



- 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: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:10-30 times faster than process creation
- Better scalability for multiprocessor architecture

- 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: 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: 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
  - New threads service each request: one thread per request
  - The main process create these threads

Thread example: 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.

- 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: management done by user-level threads library
    - A few well extablished primary thread libraries
    - POSIX Pthreads, Windows threads, Java threads

User threads and kernel threads

- User threads: management done by user-level threads library
  - A few well extablished primary thread libraries
  - POSIX Pthreads, Windows threads, Java threads
- Kernel threads - Supported by the Kernel
  - Exists virtually in all general purpose OS
  - Windows, Linux, Mac OS X

User threads and kernel threads

- User threads: management done by user-level threads library
  - A few well extablished primary thread libraries
  - POSIX Pthreads, Windows threads, Java threads
- Kernel threads - Supported by the Kernel
  - Exists virtually in all general purpose OS
  - Windows, Linux, Mac OS X

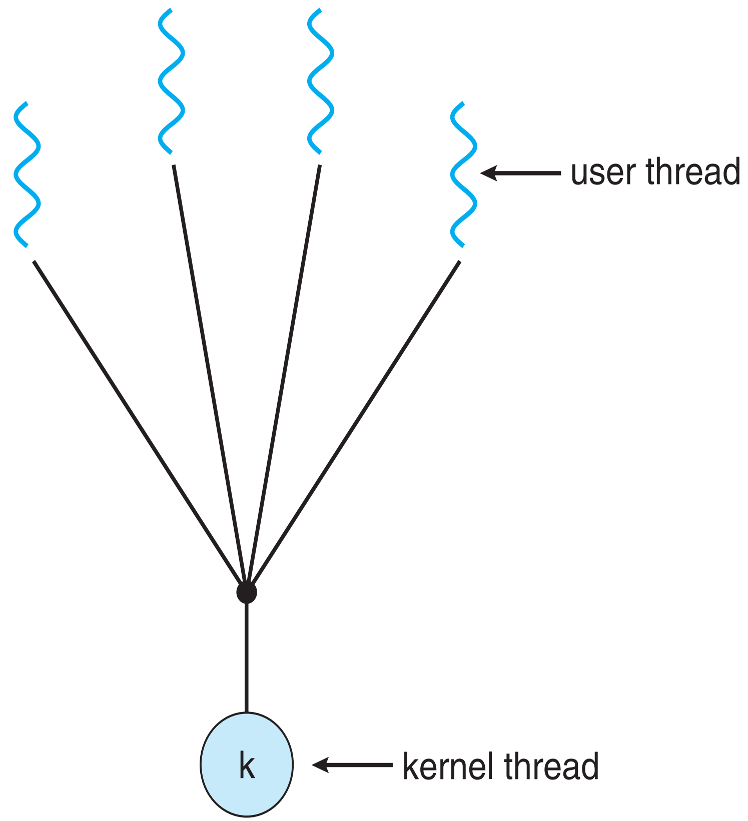
As you might have guessed: Even user threads will ultimately need kernel thread support

Multithreading Models

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

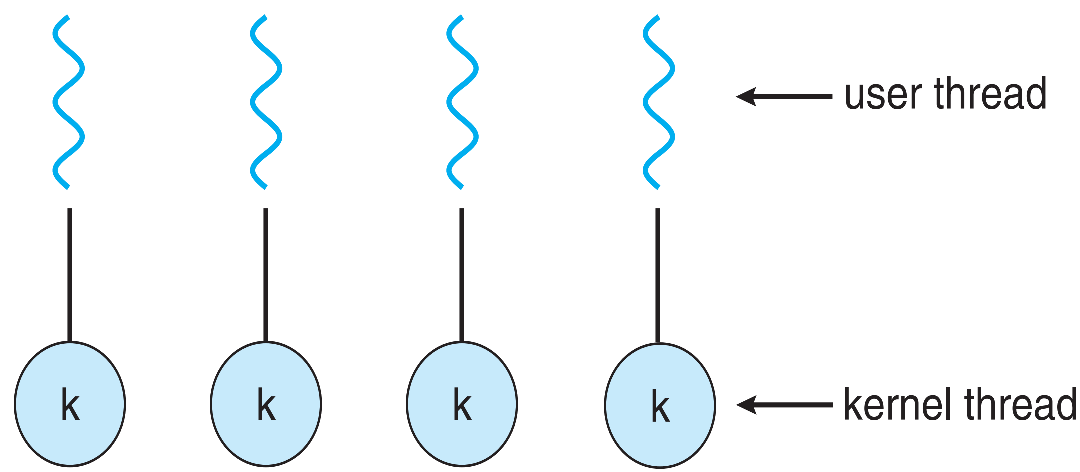
Many-to-One

- Many user-level threads mapped to single kernel thread
- Blocking one thread causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Old model: Only few systems currently use this model



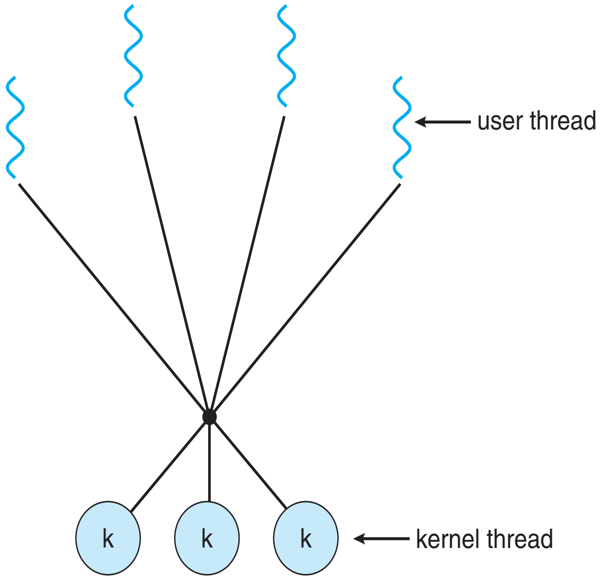
One-to-One

- Each user-level thread maps to one kernel thread
- A user-level thread creation -> a kernel thread creation
- More concurrency than many-to-one
- #threads per process sometimes restricted due to overhead on kernel
- Windows. Linux



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create sufficient number of kernel threads
- Windows with the *ThreadFiber* package



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

POSIX Pthread: basics

- May be provided either as user level or kernel level library
- Global data: Any variable/data declared globally are shared among all threads of the same process
- Local data: Data local to a function (running in a thread) are stored in thread stack
- Example:
    - A separate thread in created that calculates the sum of N natural numbers (N is an input)
    - The parent thread waits for the child thread to end

Now the code

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

Now the code

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

int sum; // data shared over threads

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

Now 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){





}
```

Now 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



}

void *runner (void *param){





}
```

Now 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){




}
```

Now 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

}

void *runner (void *param){




}
```

Now 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){




}
```

Now 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;



}
```

Now 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;}

}
```

Now 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){
…
}
```

Next class

- More on POSIX Pthread library
  - Thread scheduling
  - Thread cancellation
  - Signal handling