Operating Systems

Dr. Rana Asif Rehman

CS2006

Lecture 9

Threads

1st March 2023

1 CS-2006 Operating Systems

1

Background

Basic concepts

 Multiprogramming: Many processes one CPU. Interleaving between multiple processes. Process relinquish CPU itself.

 Multiprocessing: refers to more than one CPU. Systems can be both multiprogramming and multiprocessing at the same time.

 Multitasking: refers to modern operating systems doing preemption for switching processes/threads. OS gives a time quantum to each process/thread.

 Time sharing: When CPU time is being shared between processes, it is called time sharing system. Multiprogramming & multitasking systems are time sharing systems.

 Multithreading: is an execution model that allows a single process to have multiple code segments (threads) that run concurrently within the context of that process.

3

Motivation

 Most modern applications are multithreaded Threads run within application  Multiple tasks with the application can be

implemented by separate threads  MS Word

 One thread for spell checking

 One thread for output of keystrokes  Process creation is heavy-weight while thread creation is light-weight

 Increases efficiency

 Kernels are generally multithreaded

4

Introduction

 Each process has

1. Own Address Space

2. Single thread of control

 A process model has two concepts: 1. Resource grouping

2. Execution

 Sometimes it is useful to separate them5 CS-2006 Operating Systems

5

Unit of Resource Ownership

 A process has an

 Address space

 Open files

 Child processes

 Accounting information

 Signal handlers

 If these are put together in a form of a process, can be managed more easily

6 CS-2006 Operating Systems

6

Unit of Dispatching

 Path of execution

 Program counter: which instruction is running

 Registers:

 holds current working variables

 Stack:

 Contains the execution history, with one entry for each procedure called but not yet returned

 State

 Processes are used to group resources together  Threads are the entities scheduled for execution on the CPU Threads are also called lightweight process (LWP)

7 CS-2006 Operating Systems

7

Its better to distinguish between the two conceptsUnit of Resource Unit of Dispatch

s

s

e

c

o

r

p

t

h

g

i

e

w

y

v

a

e

H

Address

Address

space/Global

Variables

Open files

Child processes Accounting info Signal handlers Program counter Registers

In case of multiple

Stack

threads per process

State

Split

Program

Counter

space/Global Variables Open files

Program

Registers Counter

Stack

Registers Program

State

Child processes Accounting Info

Stack

Counter State

Share

Signal Handlers

8 CS-2006 Operating Systems

RegistersStack

State

8

Multithreaded Server Architecture

9

What are threads?

Thread

 A thread is a flow of execution through the process code.  A thread will always belong to some process.  A process can have multiple threads but starts with one only Threads share:

 Same data as that of its process

 Same code as that of its process

 Same files as that of its process

 Each thread has its own

 Register.

 Stack.

 Program Counter.

11

Single and Multithreaded Processes

|  |
| --- |

12

Benefits

 Responsiveness – may allow continued execution if part of process is blocked, especially important for user interfaces

 Resource Sharing – threads share resources of process, easier than shared memory or message passing

 Economy – cheaper than process creation, thread switching lower overhead than context switching

 Scalability – process can take advantage of multiprocessor architectures

13

Process vs. Thread

 Process is heavy weight or

 Thread is light weight, taking

resource intensive.

 Process switching needs interaction with operating

lesser resources than a process.

system.

 Thread switching does not need to interact with operating system.

 In multiple processes each

process operates

 One thread can read, write or

independently of the others.

change another thread's data.

14

Thread Properties

Multicore Programming

 Multicore or multiprocessor systems offer real speed up if multiple threads are used

 Programmers must accurately use threads  Divide independent problems and assign to threads  Keep a balance in dividing activities, do not overburden one or more threads

 Split Data too, so threads can work in parallel

 Data should be split properly, so that dependency among threads remains minimum  Testing and debugging of the multi threaded applications is difficult

 Parallelism implies a system can perform more than one task simultaneously  Concurrency enables more than one task making progress

16

Concurrency vs. Parallelism Concurrent execution on single-core system:

 Parallelism on a multi-core system:

17

 Types of parallelism

 Data ~~parallelism~~ – ~~distributes subsets of thesame data across multiple cores, same operation~~~~on each~~  Task ~~parallelism~~ – ~~distributing threads across~~~~cores, each thread performing unique operation~~

Multicore Programming (Cont.)  As # of threads grows, so does architectural

support for threading

 CPUs have cores as well as hardware threads

 Consider Oracle SPARC T4 with 8 cores, and 8hardware threads per core 18 CS-2006 Operating Systems

18

 User threads - management done by user-level threads library Kernel threads - ~~Operating System managed threads acting on kernel~~

 Examples – virtually all general purpose operating systems, including:

User Threads and Kernel Threads

 The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts  Three primary thread libraries:

 POSIX Pthreads

 Windows threads

 Java threads

 Windows

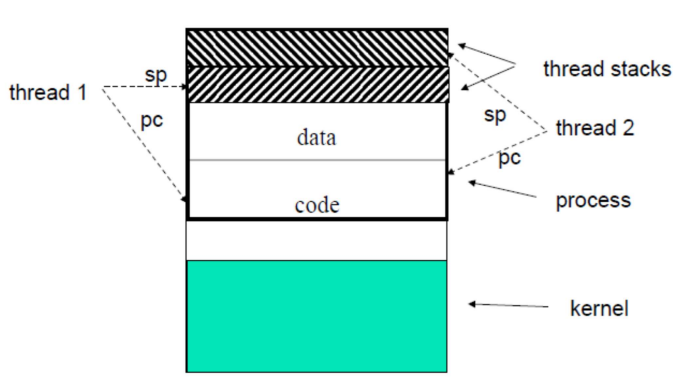
 Solaris

 Linux

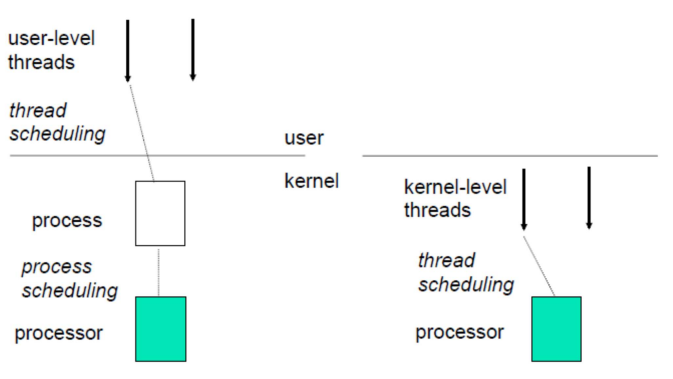
 Tru64 UNIX

 Mac OS X

19

User-Level Thread Implementation20 CS-2006 Operating Systems

20

User-Level vs. Kernel-Level Threads21 CS-220 Operating Systems

21

User-level threads vs. kernel-level threads

 User-level threads are faster to

 Kernel-level threads are slower

create and manage.

to create and manage.

 Implementation is by a thread

 Operating system supports

library at the user level.  OS doesn’t recognize user

creation of Kernel threads.

level threads.

 OS recognizes kernel-level threads

 Multi-threaded applications

 Kernel routines themselves

cannot take advantage of

can be multithreaded.

multiprocessing  If one user level thread

 If one kernel thread perform blocking operation then

perform blocking operation then entire process will be blocked.

another thread can continue execution.

22

Multithreading Models

User-level threads can be implemented using any of the three models:

 Many-to-One

 One-to-One

 Many-to-Many

23

~~parallel on muticore system because~~

Many-to-One

 Synonym of user-level threads  Many user-level threads mapped to single kernel thread

 One thread blocking causes all to block

 Multiple threads may not run in

only one may be in kernel at a time  Few systems currently use this

model

 Examples:

 Solaris Green Threads

 GNU Portable Threads

24

One-to-One

 Creating a user-level thread creates a kernel thread More concurrency than many-to-one  One user-level thread is mapped onto one kernel level thread.

 Even if one thread is blocked, others will still run  Allows multiple threads to run on multiprocessor system Number of threads per process sometimes restricted due to overhead  Examples

 Windows

 Linux

 Solaris 9 and later

25

Many-to-Many Model

 Allows many user level threads to be mapped to many kernel threads  The number of kernel level threads maybe equal to or less than user

level threads

 Allows the operating system to

create a sufficient number of

kernel threads

 Many-to-one: suffered from lack of concurrency

 One-to-one: suffered from

overhead of too many kernel level

threads

 Many-to-many suffers from none

26

Threads In Action

C function call vs. thread creation

int x; ---

int x; ---

12

12

---

function()

3

4

---

pthread\_create() 2

28

Creating a new thread

int pthread\_create(

pthread\_t \*thread,

const pthread\_attr\_t \*attr,

void \*(\*start\_routine) (void \*), void \*arg); int pthread\_join(

pthread\_t thread, // thread to join void \*\*value\_ptr // store value returned by thread);

int pthread\_exit (void \*retval); int pthread\_attr\_init(pthread\_attr\_t \*attr); int pthread\_attr\_destroy(pthread\_attr\_t \*attr); unsigned int sleep(unsigned int seconds);

29

Practical work (explanation) pthread\_create(pthread\_t\* , NULL, void\*, void\*)

 First Parameter is pointer of thread ID it should be different for all threads.

 The second argument points to a pthread\_attr\_t structure whose contents are used at thread creation time to determine attributes for the new thread. Null means use default attr.

 Third parameter is address of function which we are going to use as thread.

 Forth parameter is argument to function. Only one is allowed.

30

Practical work (explanation)

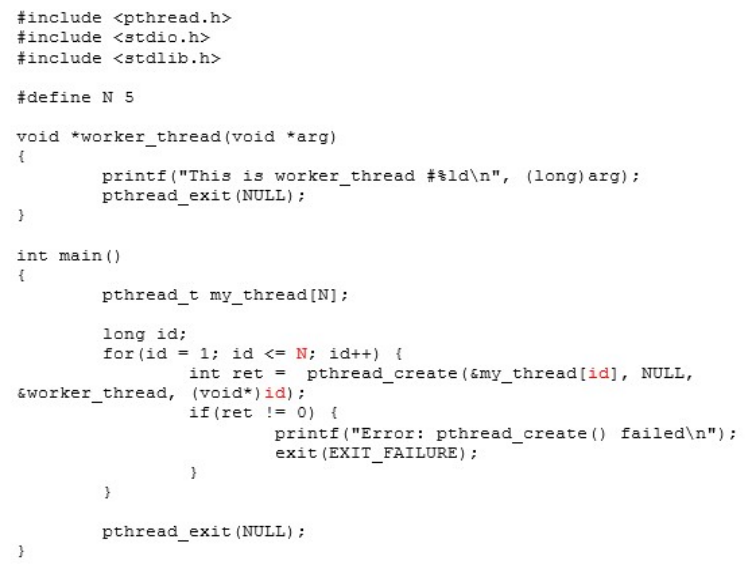
pthread\_join(i pthread\_t , void\*\*)

Pthread join is used in main program to wait for the end of a particular thread.

 First parameter is Thread ID of particular thread

 Second Parameter is used to catch return value from thread.

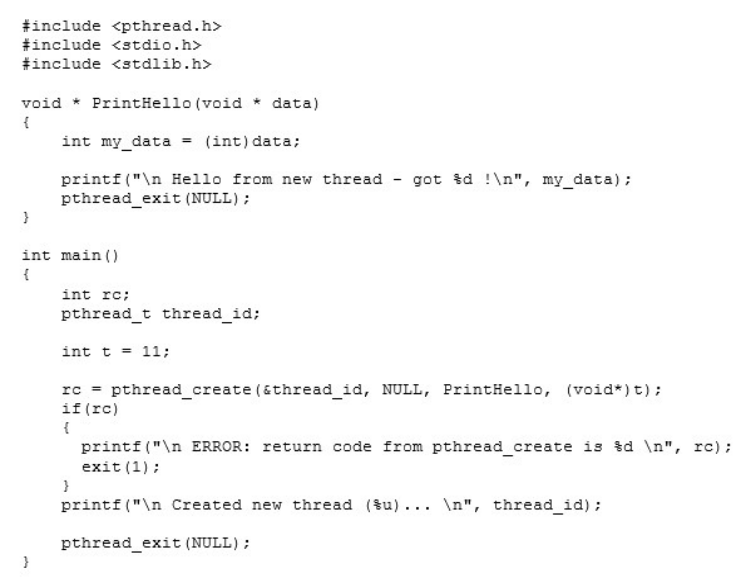
31

32

Output

This is worker\_thread #1 This is worker\_thread #2 This is worker\_thread #3 This is worker\_thread #4 This is worker\_thread #5

33

34

~~Hello from new thread - got 11~~

Output

Created new thread (4) ...

35

Questions

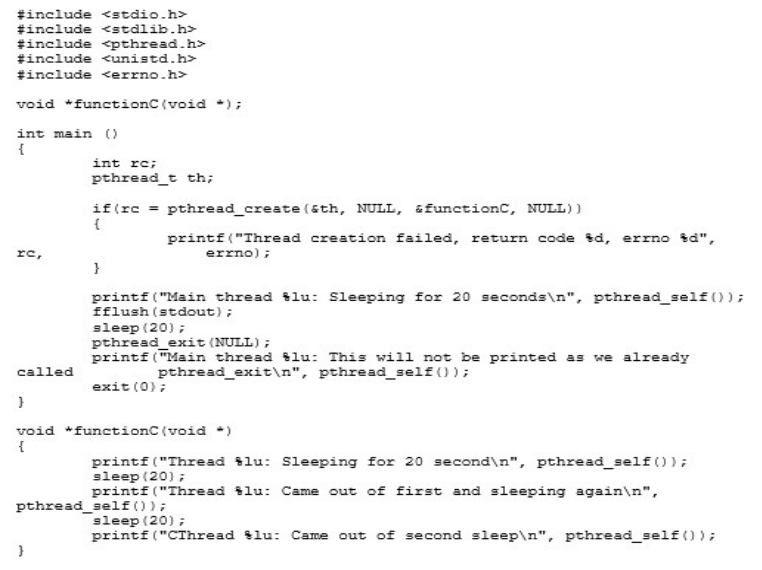
 What happens when a thread calls an exit() function?

 Can a function that is a part of the program be called from a thread?

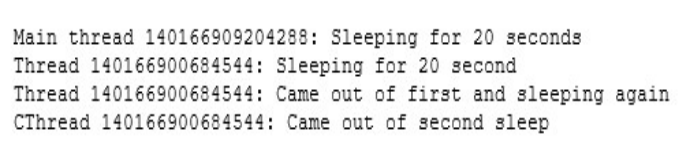
 What happens when you call a fork() in a thread?

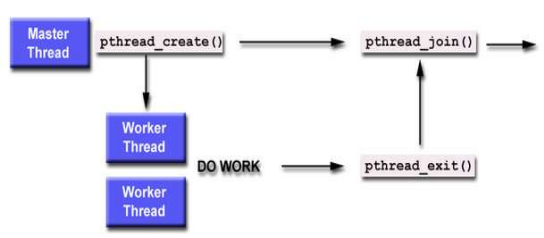
 What happens when you call pthread\_exit() in the main thread?

36

37

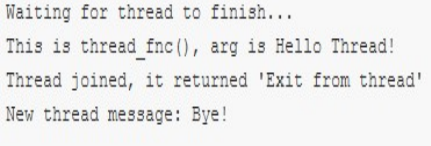
Output

38

39

40

Output

41

Practice Problem

Write a program in which you are going to create a thread. The main thread will pass an integer value N to the newly spawned thread. This thread will sum the numbers starting from 1 till the number N. The main thread will display the updated sum. Remember that the main thread will have to wait for the spawned thread to complete its work. SKELETON:

int sum;

void \*runner(void \*param);

main()

{

pthread\_create(tid, attr, runner, int\_arg); Pthread\_join(tid);

}

void \* runner(void \*param)

{

sum…

pthread\_exit(sum);

}

42

pthread\_t tid; /\* the thread identifier \*/

Thread example

pthread\_attr\_t attr; /\* set of thread attributes \*/if(argc != 2){

fprintf(stderr,"usage: a.out <integer value>\n");

#include <pthread.h>

#include <stdio.h>

int sum; /\* this data is shared by the thread(s) \*/

void \*runner(void \*param); /\* threads call this function \*/

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

return -1;

}

43

Thread example cont.

if (atoi(argv[1]) < 0) {

fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));return -1;

}

/\* get the default attributes \*/ pthread\_attr\_init(&attr); /\* create the thread \*/

pthread\_create(&tid,&attr,runner,argv[1]);/\* wait for the thread to exit \*/ pthread\_join(tid,NULL);

printf("sum = %d\n",sum); }

44

int ~~i, upper = atoi(param);~~ for (i ~~= 1; i <= upper; i++)~~

Thread example cont

/\* The thread will begin control in this function \*/void \*runner(void \*param) {

sum = 0;

sum += i;

pthread\_exit(0);

}

How to run: ./program 5

Expected output: 15

45

Practice

 Write a program that gets the following problem solved in two threads.

 (3 \* 10) + (5 \* 4)

 First half via thread 1 and second half via thread 2

 Write the program in which you will search for the two halves of an int array for a value, each half using one thread.

46

Non-deliverable assignment  Write a program in which you create a thread that calculates the factorial of a number n that is input by a user. The thread attributes must be changed are:  Scope

 Scheduling priority

 Write a program that prints the first 15 numbers(starting from 1-15) each after a delay of one second using threads  Why do we fork when we can thread?

47

~~//agruments that we pass in the function~~  // we create struct if we have to pass multiple argumentsstruct arg\_struct {

Solution Practice example\_1 1/5int arg1;

#include <iostream>

int arg2;

#include<pthread.h>

#include<stdio.h>

using namespace std;

#include <string.h>

};

48

{~~struct arg\_struct \*args = (struct arg\_struct \*)arguments;~~ int \*new\_ptr=new int;

Solution Practice example\_1 2/5

//function that we use in threads

void\* myFunc(void\* arguments)

\*new\_ptr=args->arg1\*args->arg2;

pthread\_exit( (void\*) new\_ptr);

//do not use exit routine, it will terminate the whole process}

49

int main()

~~pthread\_t thread1\_id,thread2\_id;~~

struct arg\_struct thread1\_var;

Solution Practice example\_1 3/5struct arg\_struct thread2\_var;

{

// creating id for thread

//initializing variables

thread1\_var.arg1=3;

thread1\_var.arg2=10;

thread2\_var.arg1=5;

thread2\_var.arg2=4;

50

Solution Practice example\_1 4/5

// creating threads

if (pthread\_create(&thread1\_id, NULL, &myFunc,(void \*) &thread1\_var)==-1)

{

cout<<"Thread Creation Failed!"<<endl;

return 1;

}

if (pthread\_create(&thread2\_id, NULL, &myFunc,(void \*) &thread2\_var)==-1)

{

cout<<"Thread Creation Failed!"<<endl;

return 1;

}

51

~~int \*ptr\_t1\_ret,\*ptr\_t2\_ret;~~

Solution Practice example\_1 5/5

// variables to store the return value

// wait for the end of a particular thread pthread\_join(thread1\_id, (void\*\*) &ptr\_t1\_ret);pthread\_join(thread2\_id, (void\*\*) &ptr\_t2\_ret);// display result

cout<<\*ptr\_t1\_ret + \*ptr\_t2\_ret;

}

52

Example 2

53

~~//agruments that we pass in the function~~

// we create struct if we have to pass multiple arguments

Solution Practice example\_2 1/5struct arg\_struct {

int array[10]={1,2,3,4,5,6,7,8,9,10};

#include <iostream>

int size=10;

#include<pthread.h>

int search=11;

#include<stdio.h>

using namespace std;

#include <string.h>

};

54

Solution Practice example\_2 2/5

//function that we use in threads

// to search in 1st half

void\* myFunc1(void\* arguments)

{

struct arg\_struct \*args = (struct arg\_struct \*)arguments; int \*val\_p=args->array;

int \*val=new int;

int i = 0;

for( i = 0; i < args->size/2; i++){

if(val\_p[i]==args->search){

\*val=val\_p[i];

}

}

pthread\_exit( (void\*) val); }

55

Solution Practice example\_2 3/5

// to search in 2nd half

void\* myFunc2(void\* arguments)

{

struct arg\_struct \*args = (struct arg\_struct \*)arguments; int \*val\_p=args->array;

int \*val=new int;

int i = 0;

for( i = args->size/2; i < args->size; i++){ if(val\_p[i]==args->search){

\*val=val\_p[i];

}

}

pthread\_exit( (void\*) val); }

56

int main()

{~~// createing id for thread~~

~~pthread\_t thread1\_id,thread2\_id;~~

~~struct arg\_struct thread\_var;~~

Solution Practice example\_2 4/5

// creating threads

if (pthread\_create(&thread1\_id, NULL, &myFunc1,(void \*) &thread\_var)==-1){

cout<<"Thread Creation Failed!"<<endl;

return 1;

}

if (pthread\_create(&thread2\_id, NULL, &myFunc2,(void \*) &thread\_var)==-1){

cout<<"Thread Creation Failed!"<<endl;

return 1;

}

57

Solution Practice example\_2 5/5

// variables to store the return value int \*ptr\_t1\_ret,\*ptr\_t2\_ret;

// wait for the end of a particular thread pthread\_join(thread1\_id, (void\*\*) &ptr\_t1\_ret); pthread\_join(thread2\_id, (void\*\*) &ptr\_t2\_ret);

// display result

if(\*ptr\_t1\_ret>0){

cout<<"found in first half of array"; }

else if(\*ptr\_t2\_ret>0){

cout<<"found in 2nd half of array"; }

else

{cout<<"not found";}

}

58

Thanks

59