CSC 4320/6320: Operating Systems

Chapter 04: Threads and Concurrency-Part 2

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Outline

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

Objectives

- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux operating systems represent threads
- Designing multithreaded applications using the Pthreads, Java, and Windows threading APIs

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

It is possible to create a thread library without any kernel-level support.

True ✓
False

User-level Threads

- **Thread Creation**: The thread library allocates memory for the thread's stack and registers, and it creates the thread control block (TCB) to store its state.
- **Context Switching**: Kernel is unaware of the threads; the thread library must handle switching between threads by saving and restoring the register state.
- **Scheduling**: The thread library must implement its own scheduler to decide when to switch between threads.
- **Synchronization**: Since the threads are not managed by the kernel, the thread library must provide synchronization to coordinate access to shared resources.

Pthreads

- May be provided either as user-level (usually not now) or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation*** *(Operating-system designers may implement the specification)*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Linux & Mac OS X)
- Most modern operating systems, such as Linux and macOS, use **kernel-level threads** for Pthreads.

Pthreads Example

Task: construct a multithreaded program that calculates the summation up to a non-negative integer in a separate thread.

$$sum = \sum_{i=1}^{N} i$$

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.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[])
{
  pthread_t tid;  /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  /* set the default attributes of the thread */
    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);
}
```

Pthreads Example (Cont.)

- Note: You are free to use any valid C function name for the thread function.

- The thread function must return a void*.

- The thread function must take a single argument of type void*.

```
/* The thread will execute in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;
  for (i = 1; i <= upper; i++)
    sum += i;
  pthread_exit(0);
}
```

`void*` is a generic pointer, the thread function can return **any data type** by casting it to `void*`. A concrete example of how to use this generic pointer (`void *`) to return an integer is shown in the next slide.

How to use void* to return any data

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.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[]) {
  pthread_t tid;  /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  /* set the default attributes of the thread */
  pthread_attr_init(&attr);

  /* create the thread */
  pthread_create(&tid, &attr, runner, argv[1]);

   /* handle return value */
  void *return_value;
  /* wait for the thread to exit */
  pthread_join(tid, &return_value); /*use NULL (if not interested)*/

  int *result = (int *)return_value;
  printf("The returned value:%d\n", *result); /*a dummy ret value*/

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

  free(result);
  return 0;
}
```

```c
/* The thread will execute in this function */
void *runner(void *param){
  int i, upper = atoi(param);
  sum = 0;
  for (i = 1; i <= upper; i++)
      sum += i;

  /* do the following to return a local variable */
  int* result = malloc(sizeof(int));/*heap will be alive*/
  *result = 42;  /* e.g., store an integer value

  /* We can use both ways to return values */
  // return (void*)result; /*one way to return */
  pthread_exit((void *)result); /*another way to return */

}
```

Pthreads Code for Joining 10 Threads

A simple method for waiting on several threads using the `pthread_join()` function is to enclose the operation within a simple for loop

```
#define NUM THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
  pthread_join(workers[i], NULL);
```

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

DWORD: 32-bit unsigned integer.(double word)
LPVOID is equivalent to **void*** in standard C/C++. (Long Pointer to Void)

(DWORD*) Param: This part casts the generic param pointer to a pointer of type **DWORD***.

Windows Multithreaded C Program (Cont.)

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

  - Standard practice is to implement Runnable interface

Java Threads

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Creating thread and waiting are within the main function in public class

Questions

A ____ provides an API for creating and managing threads.

A) set of system calls
B) multicore system
C) thread library ✓
D) multithreading model

Pthreads refers to ____.

A) the POSIX standard.
B) an implementation for thread behavior.
C) a specification for thread behavior. ✓
D) an API for process creation and synchronization.

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- In implicit threading, creation and management of threads done by compilers and run-time libraries rather than programmers
- Five methods explored
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e, Tasks could be scheduled to run after a time delay or to run periodically
- Windows API supports thread pools:

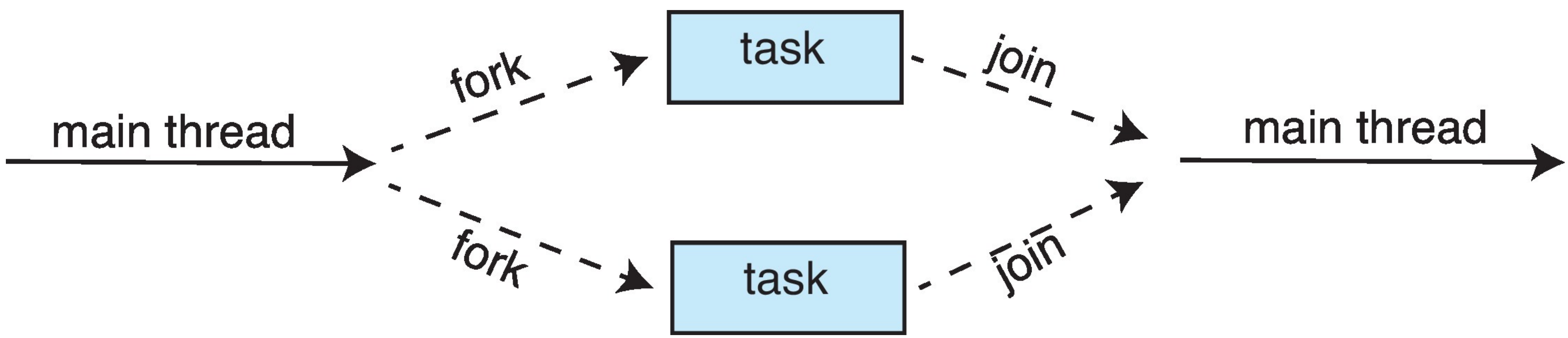
```
DWORD WINAPI PoolFunction(PVOID Param) {
   /* this function runs as a separate thread. */
}
```

How does Thread Pool work?

- When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests.
- If there is an available thread in the pool, it is awakened, and the request is serviced immediately.
- If the pool contains no available thread, the task is queued until one becomes free.
- Once a thread completes its service, it returns to the pool and awaits more work.

Fork-Join Parallelism

- Multiple threads (tasks) are **forked**, and then **joined**.



Fork-Join Parallelism

- General algorithm for fork-join strategy:

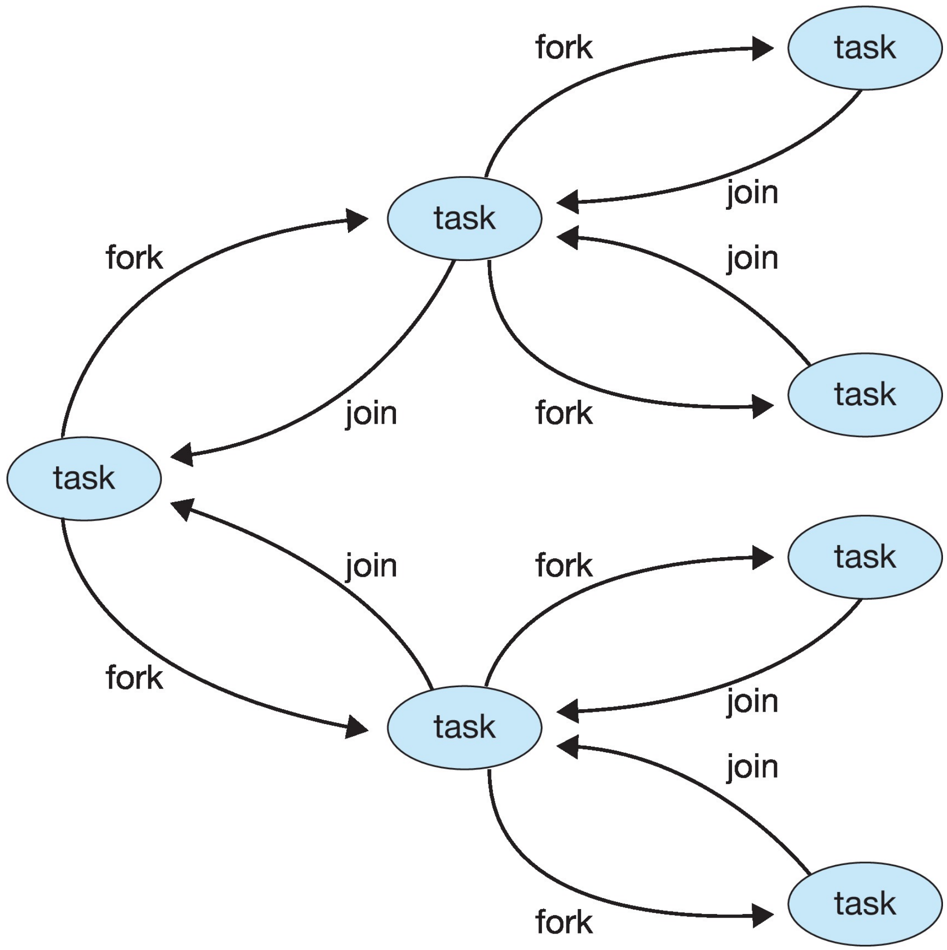
```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Java introduced a fork-join library in Version 1.7 of the API that is designed to be used with recursive divide-and-conquer algorithms such as Quicksort and Mergesort.

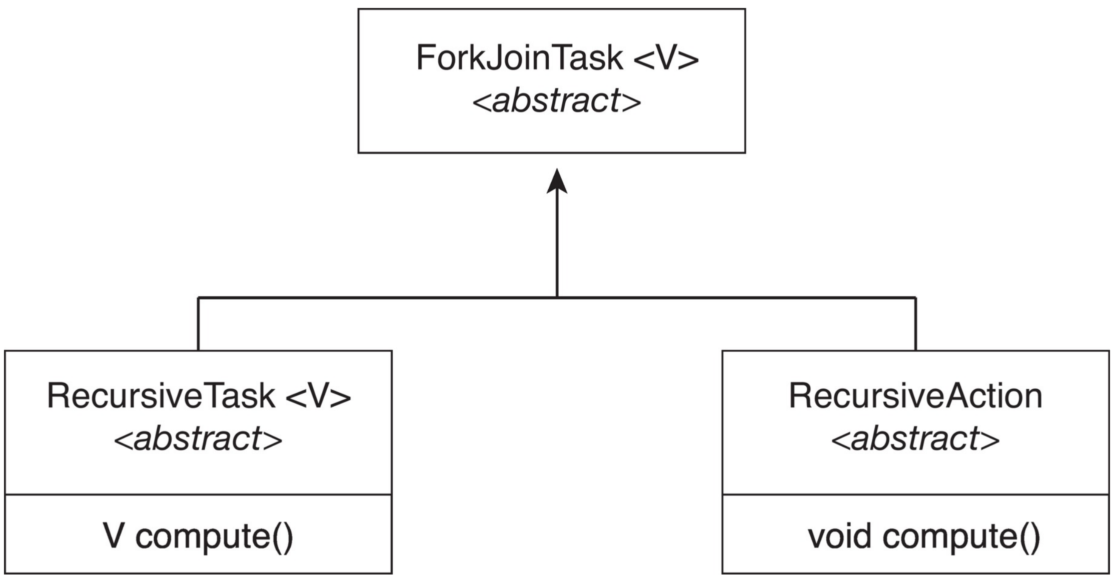
Fork-Join Parallelism



Fork-Join Parallelism in Java

- The **`ForkJoinTask`** is an abstract base class
- **`RecursiveTask`** and **`RecursiveAction`** classes extend **`ForkJoinTask`**
- **`RecursiveTask`** returns a result (via the return value from the **`compute()`** method)
- **`RecursiveAction`** used for task that does not return a result

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;
```



OpenMP

- OpenMP is a set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

`#pragma omp parallel`

Create as many threads as there are cores

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

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

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

How to execute:

```
gcc -fopenmp openmp_ex.c -o openmp
./openmp
```

OpenMP (contd.)

- Run the for loop in parallel

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

- OpenMP divides the work contained in the for loop among the threads it has created in response to the directive

```
#pragma omp parallel for
```

- Available on several open-source and commercial compilers for Linux, Windows, and macOS systems.

An Example

```c
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d\n", omp_get_thread_num());
    }
    return 0;
}
```

```
Output:
Hello from thread 2
Hello from thread 0
Hello from thread 3
Hello from thread 1
```

Grand Central Dispatch

- Apple technology for macOS and iOS operating systems
- Extensions to C, C++ and Objective-C languages, API, and run-time library
- Allows identification of parallel sections, like OpenMP
- Manages most of the details of threading
- Block is in "^{ }" :

```
^{ printf("I am a block"); }
```

- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue (i.e., takes from the queue and assigns a thread)

Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

- The same for loop written using TBB with **`parallel_for`** statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```

- TBB library will divide the loop iterations into separate "chunks" and create a number of tasks that operate on those chunks.
- Supported by major OS: Linux/Windows/Mac OS

Questions

1. A _____ uses an existing thread — rather than creating a new one — to complete a task.

A) lightweight process
B) thread pool
C) scheduler activation
D) asynchronous procedure call

B√

When OpenMP encounters the #pragma omp parallel directive, it

A) constructs a parallel region
B) creates a new thread
C) creates as many threads as there are processing cores
D) parallelizes for loops

C√

Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of `fork()` and `exec()`

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork: `fork() and vfork()`
- **`exec()`** usually works as normal – replace the running process including all threads
  - if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

Which version of `fork()` to use?

- Depends on the application.
- If exec() is called immediately after forking, then duplicating all threads is unnecessary
  - the program specified in the parameters to exec() will replace the process.
  - duplicating only the calling thread is appropriate.
- If the separate process does not call exec() after forking, the separate process should duplicate all threads.

```
pid_t pid = vfork(); /* instead of a fork() call */
```

- `vfork()` is faster than `fork()`, because memory space of the parent process is not copied.

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

An Example

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

/* define user friendly handler */
void handle_sigint(int sig) {
    printf("Caught signal %d (SIGINT)!\n", sig);
    printf("\nExiting from the program intentionally");
    exit(0);
}
int main() {
    /* set which signal we want to handle: SIGINT */
    signal(SIGINT, handle_sigint);
    printf("Press Ctrl + C (SIGINT) to test. Process ID: %d\n", getpid());
    while (1) {
        sleep(1);  /* Infinite loop */
    }
    return 0;
}
```

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

- Synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.
- Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads.

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation:** one thread terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should terminate.
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Note: Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **`static`** data
  - TLS is unique to each thread
- The gcc compiler provides the storage class keyword _thread for declaring TLS data. For example:

  - `static __thread int threadID;`

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create? (varies; good to start with creating **one LWP per available core)**
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads