Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department
1400-1 semester

Zeinab Zali

Session 8: Implicit Threading, Signals, ...

Implicit Threading

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

Client-Server Example

Remember our own example

What happens if the number of alive threads exceeds the maximum number of concurrent threads that the system can support?

How can we prevent this issue?

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 periodically
- Windows API supports thread pools:

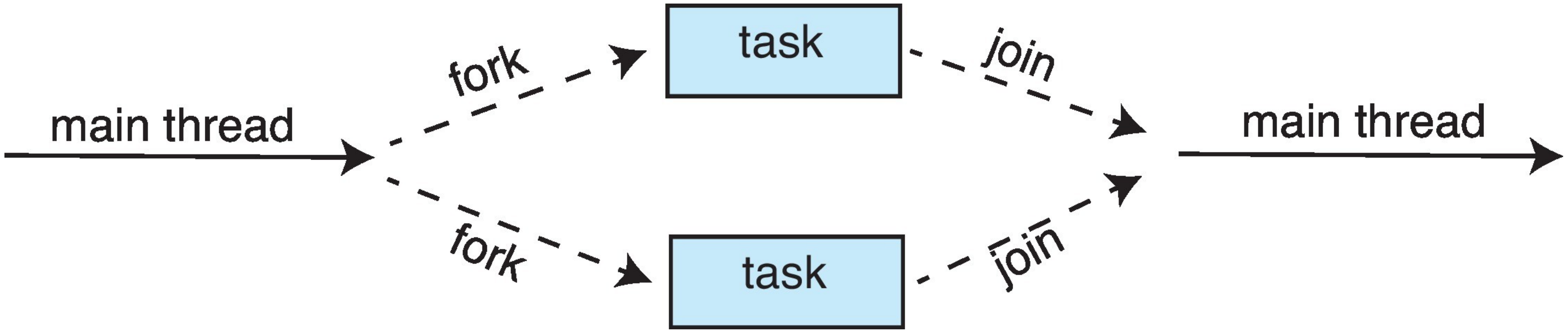
```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}
```

Sample thread pool API

```
void first_task() {…}
void second_task() {…}
void third_task() {…}
main(){
    // Create a thread pool.
    pool tp(2);
    //Add some tasks to the pool.
    tp.schedule(&first_task);
    tp.schedule(&second_task);
    tp.schedule(&third_task);
}
```

Fork-Join Parallelism

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



Fork-Join Parallelism

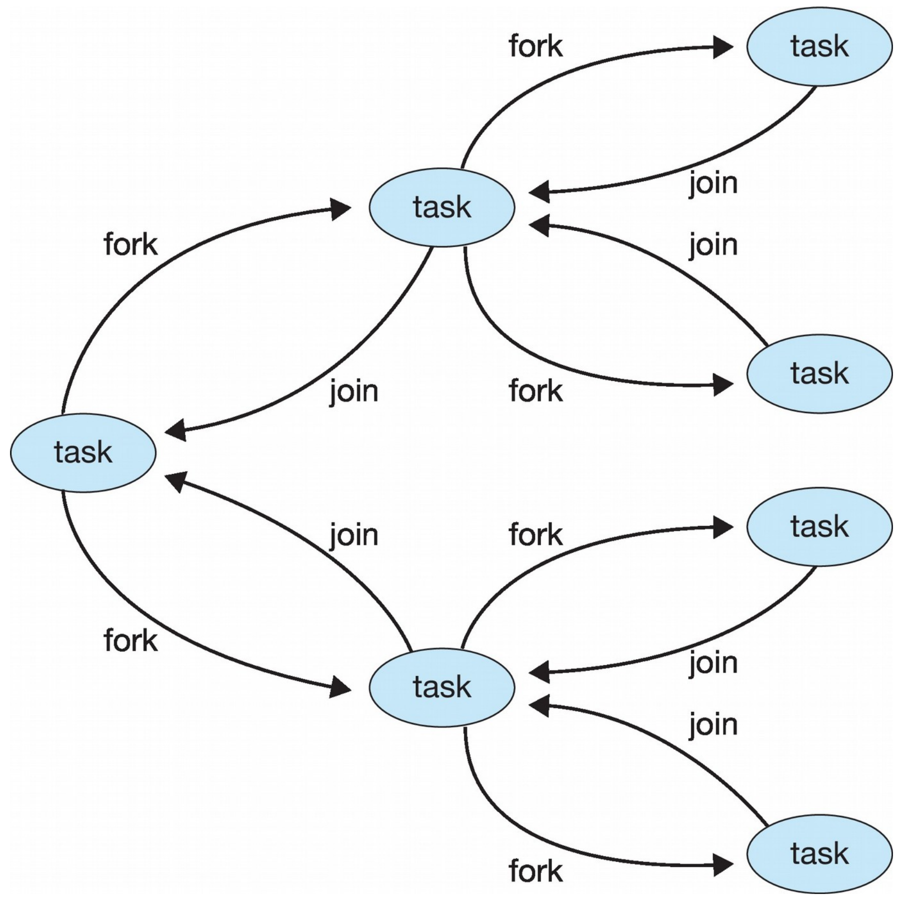
- General algorithm for fork-join strategy:

```
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
```

Fork-Join Parallelism



OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
  provides support for parallel programming
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

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

Run for loop in parallel

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIX systems have two versions of fork():
    - one that duplicates all threads
    - one that duplicates only the thread that invoked the fork() system call.
- **`exec()`** usually works as normal – replace the running process including all threads
  - So when exec() is called immediately after forking, forking only the calling thread is sufficient

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

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 processes

kill(pid_t, signal)

pthread_kill(thread_t, signal)

Thread Cancellation

- Terminating a thread before it has finished
  - Suppose multiple threads searching for a record in a database and one of them find it
  - a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be canceled

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

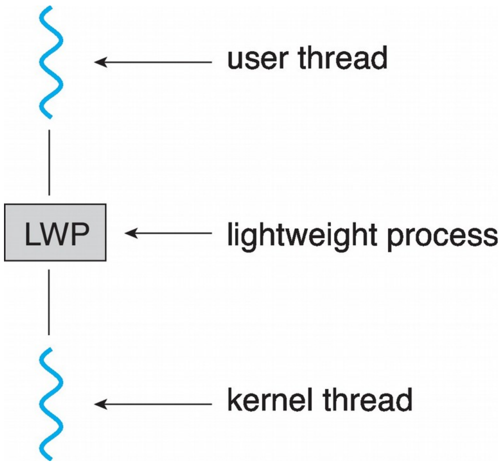
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - Ex: **`pthread_testcancel()`**
    - `Ex: read function`
- On Linux systems, thread cancellation is handled through signals

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

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?
- 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



Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)

getconf GNU_LIBPTHREAD_VERSION

End of Chapter 4