Chapter 4: Threads

CS 3423 Operating Systems

Fall 2019

National Tsing Hua University

Overview

- Introduction to Threads
- Multithreading Models
- Threaded Case Study
- Threading Issues

Objectives

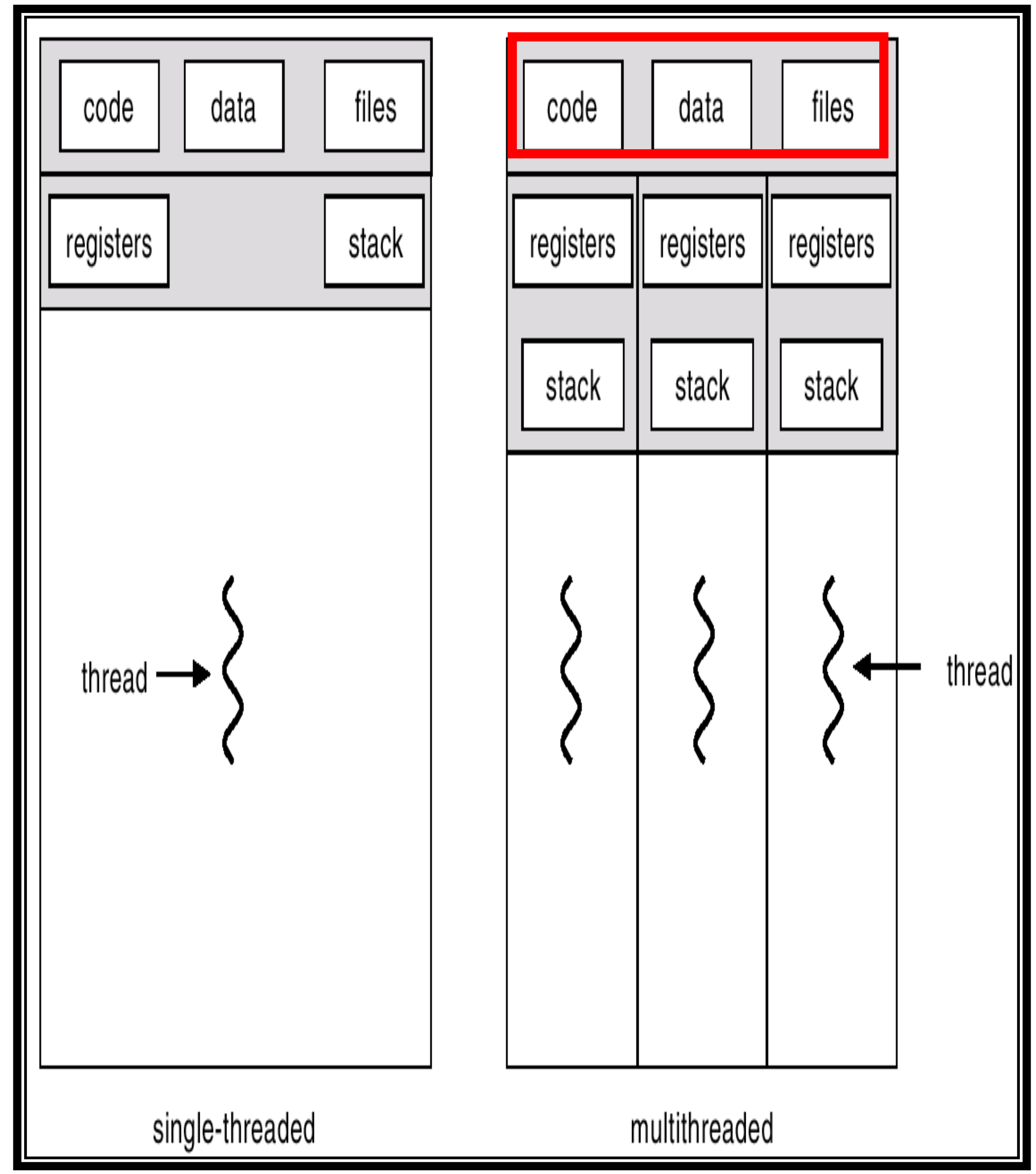
- Introduce thread as a *fundamental unit of CPU utilization*
- Discuss APIs for the Pthreads thread libraries
- Implicit threading
- Case studies of Threads Libraries and OSs

Motivation

- Multiple tasks in modern applications
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight
- Solution: thread creation is light-weight
  - Can simplify code, increase efficiency
  - Kernels are generally multithreaded

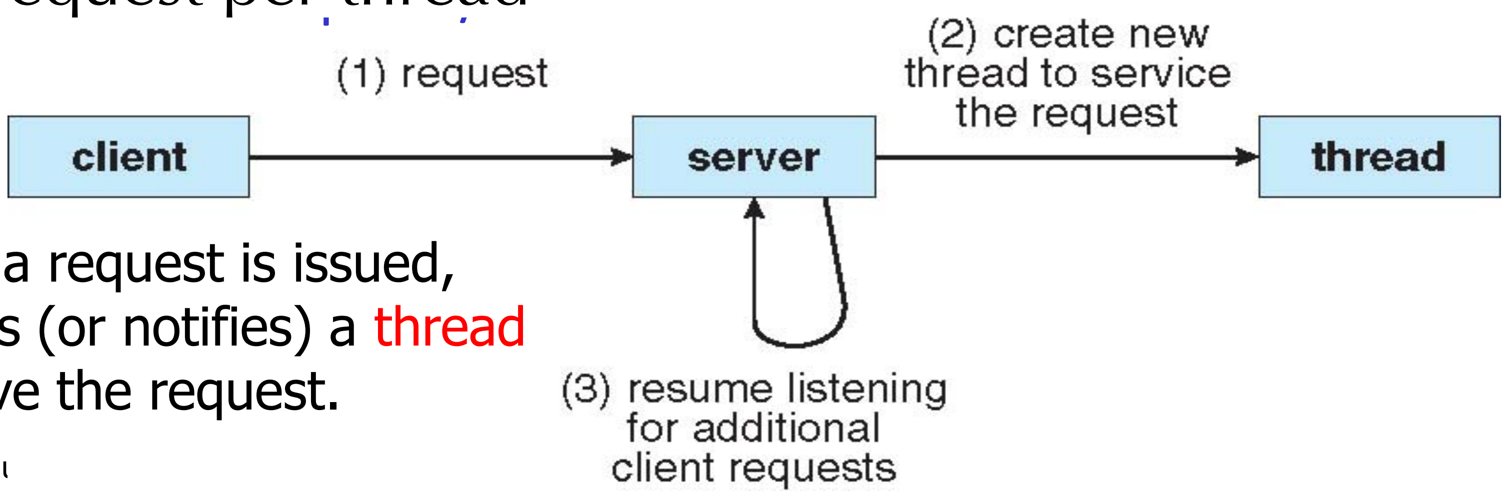
Threads

- aka lightweight process:
  - basic unit of CPU utilization
- All threads of a process share
  - code section, data section, open files, signals
- Each thread has its own
  - thread ID, program counter, register set, stack
  - *thread control block* can be used to save thread state, analogous to PCB for processes



Examples

- web browser
  - one thread displays content
  - another thread receives data from network
- web server
  - spawn one process per request => too heavyweight
  - use threads => lighter weight, better sharing of code and resources
- RPC server
  - one RPC request per thread



When a request is issued, creates (or notifies) a thread to serve the request.

Benefits of Multithreading

- Responsiveness
  - one thread blocked, another thread may perform a lengthy operation
- Resource Sharing
  - several threads run in the same address space, easier sharing than interprocess shared memory or message passing
- Economy
  - process-level operation is heavyweight
  - Solaris: process creation is 30x as slow as thread creation
    context switch with process is 5x slower than thread switching
  - Threads: switch register set but not memory management
- Scalability
  - threads may run in parallel on multiprocessor

Why Threads

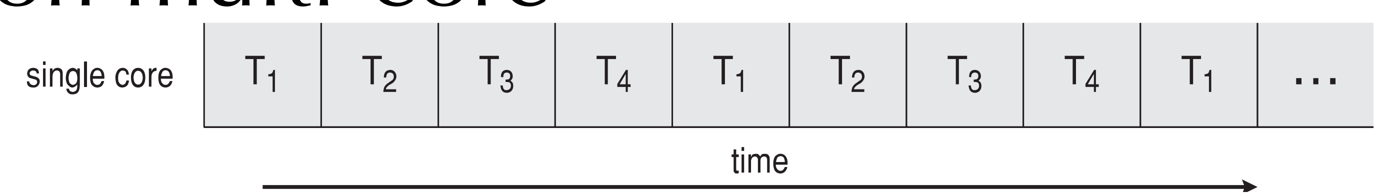
Platform\ Op	Creation			Communication		
	fork()	pthread create()	speed up	MPI shared (GB/s)	Pthread Mem-CPU	speed up
AMD 2.4 GHz Opteron	17.6	1.4	15.6x	1.2	5.3	4.4x
IBM 1.5 GHz POWER 4	104.5	2.1	49.8x	2.1	4	1.9x
Intel 2.4 GHz Xeon	54.9	1.6	34.3x	0.3	4.3	14.3x
Intel 1.4 GHz Itanium2	54.5	2.0	27.3x	1.8	6.4	3.6x

Challenges in Multicore Programming

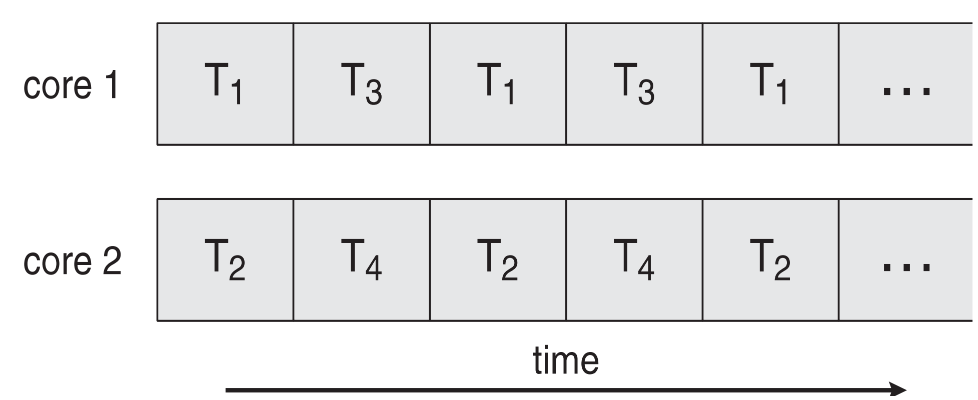
- Computation partitioning
  - into concurrent tasks
- Balancing
  - evenly distribute tasks to cores
- Data splitting
  - data units to expose data parallelism
- Data dependency
  - synchronize data accesses
- Testing and debugging

Concurrency vs. Parallelism

- Concurrency
  - multiple tasks **active** at the same time
  - one running at a time on single-core system
  - may run in parallel on multi-core

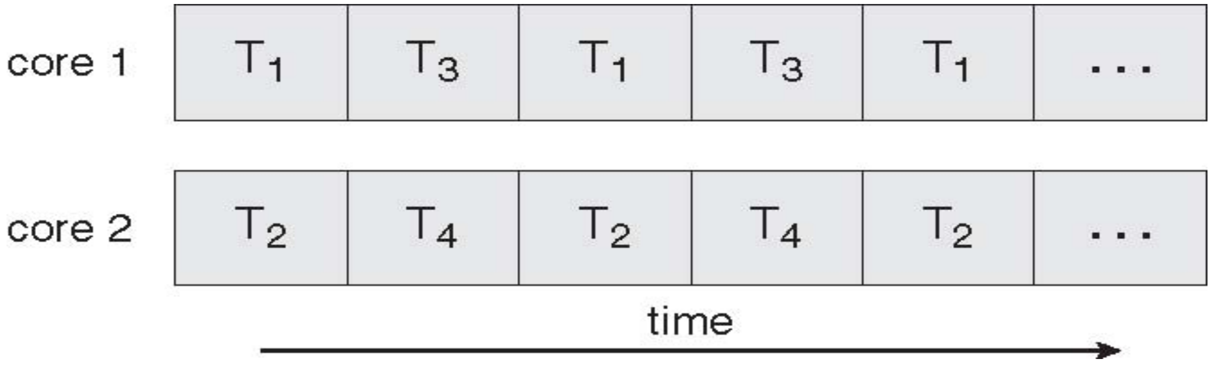


- Parallelism
  - running multiple tasks simultaneously
  - requires a multi-core system



Multicore Programming

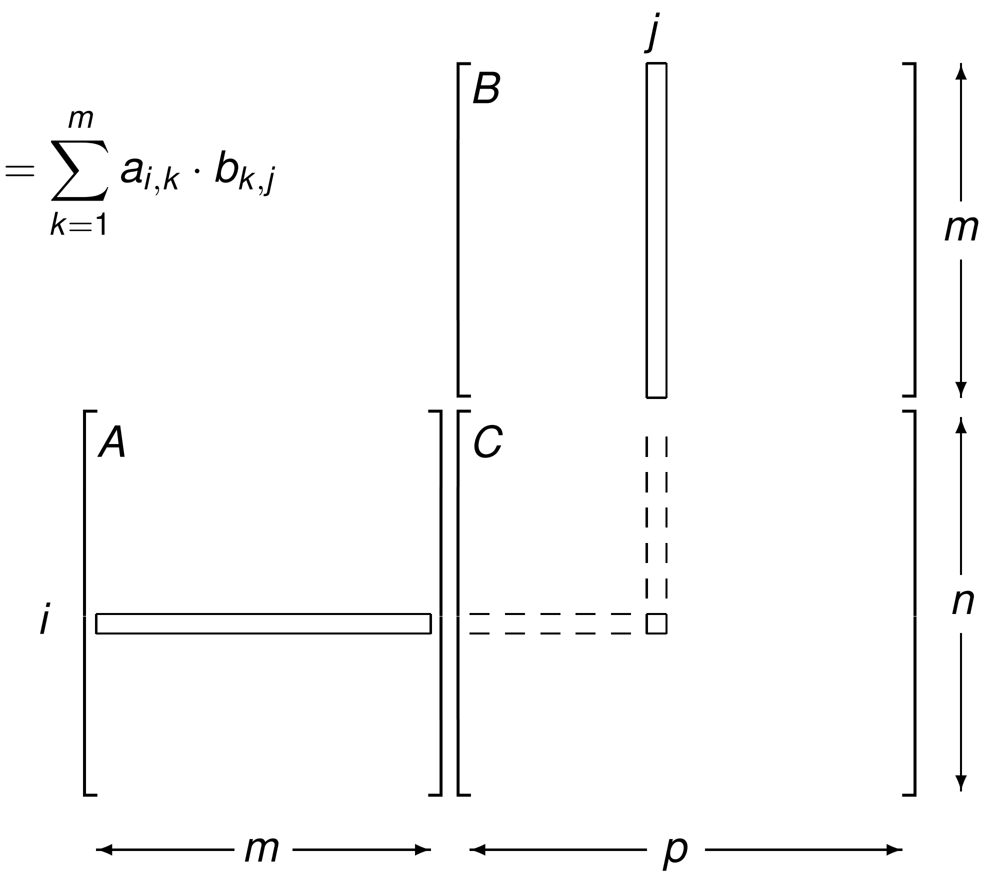
- Multicore architectures
  - Cores can share same (physical) memory
- Each core could support multiple hardware threads
  - SMT (simultaneous multithreading) architectures, e.g., Intel Hyperthreading
- Multithreading good match with multicore
  - Parallelism: threads can run in parallel if OS schedules them on multiple cores
  - data parallelism vs task parallelism



Data Parallelism

- same task running on different data
  - data may be segmented or multiple streams
  - different parts can be processed in parallel
- Examples:
  - matrix multiply
  - dot-products
    are data-parallel!

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$$



Task Parallelism

- the problem can be decomposed
  - into threads that have little mutual dependency
  - each thread runs (potentially) different code
- Examples
  - servers that serve a variety of requests
    - `http`, `ftp`, cloud drive, streaming, ...
  - Multimedia, games: audio, graphics, networking
    - but... only up to a limit (e.g., frame), and they need to synchronize ("fork-join" parallelism)

Pipeline Parallelism

- Divide a task into stages
  - Each stage is executed on its own processor
  - Assuming data is streamed
- Example: 3D Graphics pipeline for gaming
  - application (character action, game rules)
  - geometry (lighting, projection, clipping, viewport)
  - rasterization (hidden surface removal, texture, shading, alpha blending / antialiasing)
- One stage depends on previous stage for input

Series-Parallel parallelism

- Also called fork-join parallelism
  - program starts out serial
  - can spawn threads ("fork") to do work concurrently
  - threads synchronize ("join") after they finish
  - program executes in series for a while, then fork...
- Common for recursive algorithms
  - "divide-and-conquer": MergeSort, QuickSort, etc.
  - supported as "fork-join" constructs by some languages or threads packages

example: MergeSort

- MergeSort(A[])
  - Divide A into two halves L, R } executes in series
  - MergeSort(L)
  - MergeSort(R) } can run in parallel! data do not overlap
  - # conquer
    A = Merge(L, R) } executes in series (linear time)

User thread vs. Kernel thread

- User threads
  - thread management done by user-level thread library
  - OS only sees processes; does not "see" user threads
  - example: POSIX pthreads, Win32 threads, Java threads, Python threads
- Kernel threads
  - managed by the OS kernel directly
  - does not mean "threads that run in kernel mode"! (they could, but could switch to user mode to run the process)
  - e.g.: Windows 2000 (NT), Solaris, Linux, Tru64 Unix, macOS

User thread vs. Kernel thread

- User thread library
  - supports thread creation, scheduling, deletion
  - Generally fast to create and manage
  - If kernel is single threaded, when a user thread blocks => entire process blocks, even if some threads are ready to run
- Kernel threads
  - kernel performs thread creation, scheduling, etc
  - Generally slower to create and manage
  - if a thread is blocked, the kernel can schedule another thread to run

Multithreading Models

- Different ways of mapping user threads to kernel threads
- Three combinations
  - Many-to-one
  - One-to-one
  - Many-to-many
- Preemption
  - cooperative vs. preemptive

Many-to-One

- Many user-level threads mapped to one kernel thread
    - for systems that don't support kernel threads, so the process itself is “single-threaded”
    - Examples: Solaris Green threads, GNU portable threads
- Pro
    - All thread management is done in user space => efficient
- Con
    - if one user thread makes a blocking system call => whole process blocks
    - can’t run multiple such threads in parallel on multiprocessors => few systems currently use this model, as multicore is norm

One-to-One

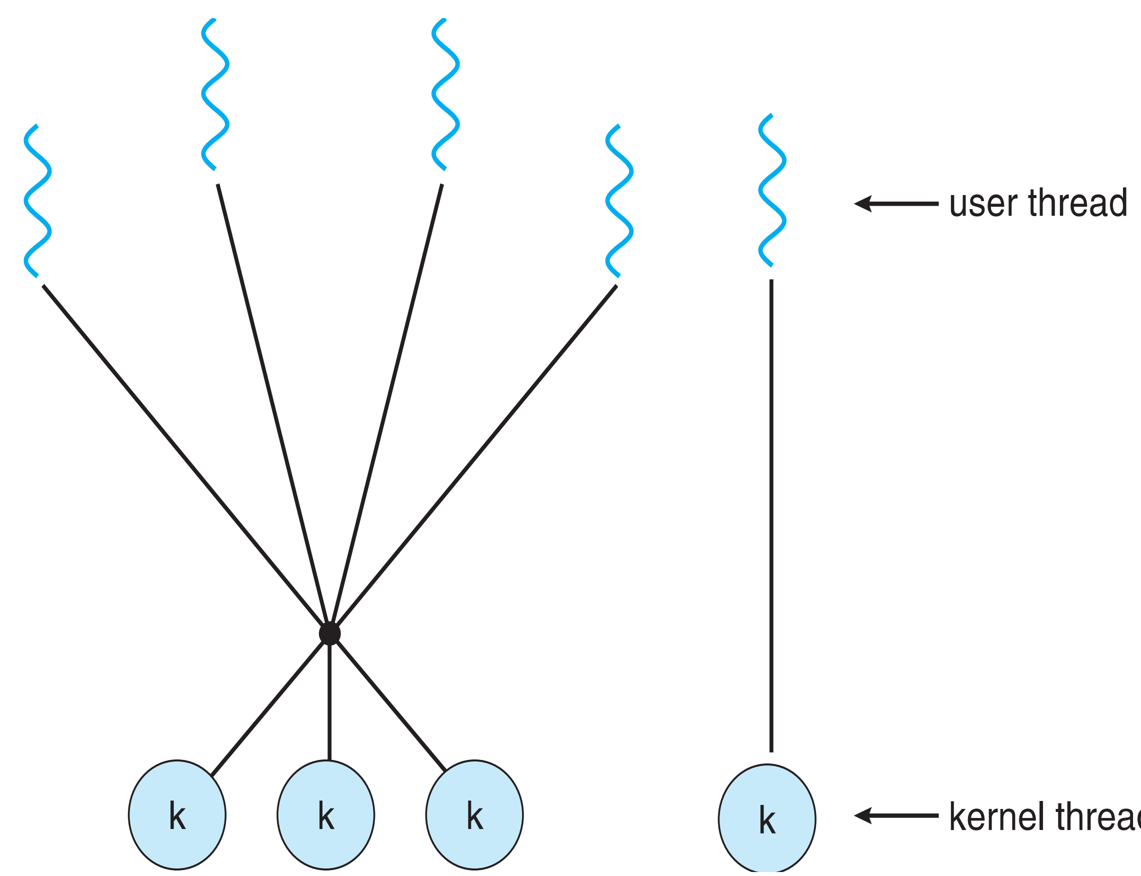
- Each user-level thread maps to a kernel thread
  - there may be a limit on the number of kernel threads
- Pro
  - More concurrency than sharing one kernel thread
- Con
  - higher overhead: each user thread is one kernel thread
- Examples
  - Windows XP/NT/2000, Linux, Solaris 9 and later
- Most popular model - for now
  - more cores now, balances between complexity and performance gain

Many-to-Many

- Map multiple user threads to a number of kernel threads
  - Some user threads may share a kernel thread
  - Developers can create as many user threads as Many-to-one
- Pro
  - threads mapped onto different kernel threads can run in parallel on a multiprocessor
  - If a user thread blocks on a call, the kernel can schedule another kernel thread for other threads of that process

Two-level Model of threads

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Review (1)

- Benefits of multithreading
  - Responsiveness, economy, resource utilization and sharing
- Types of parallelism
  - data parallelism, task parallelism
- Challenges of multithreading programming
- User threads vs Kernel threads
- Thread models
  - many-to-one, one-to-one, many-to-many

Thread concepts

- main thread
    - the initial thread of control that already exists and running when the program starts
    - main thread creates other threads
- worker thread
    - created thread, maybe ready to accept work or is working
- thread pool
    - pool of worker threads ready to accept work

Thread Primitives (1/3)

- ***create*** a thread; aka ***spawn*** a thread
  - create a thread to run a function
    instead of cloning the creator's thread
  - may start running automatically,
    or may need to call a start() explicitly to run the created thread
  - No parent-child relationship like fork()!

Thread Primitives (2/3)

- ***join*** a thread *t*
  - creator waits for a thread *t* to finish (if not already), then release its resources
  - somewhat like calling wait() on a child process.
- ***detach*** a thread *t*
  - creator tells threads manager to automatically release thread *t*'s resources when it finishes,
  - otherwise, after *t* finishes, its resources won't be released until creator calls join(*t*)
  - once someone detaches *t*, can't join *t* any more!!

Thread Primitives (3/3)

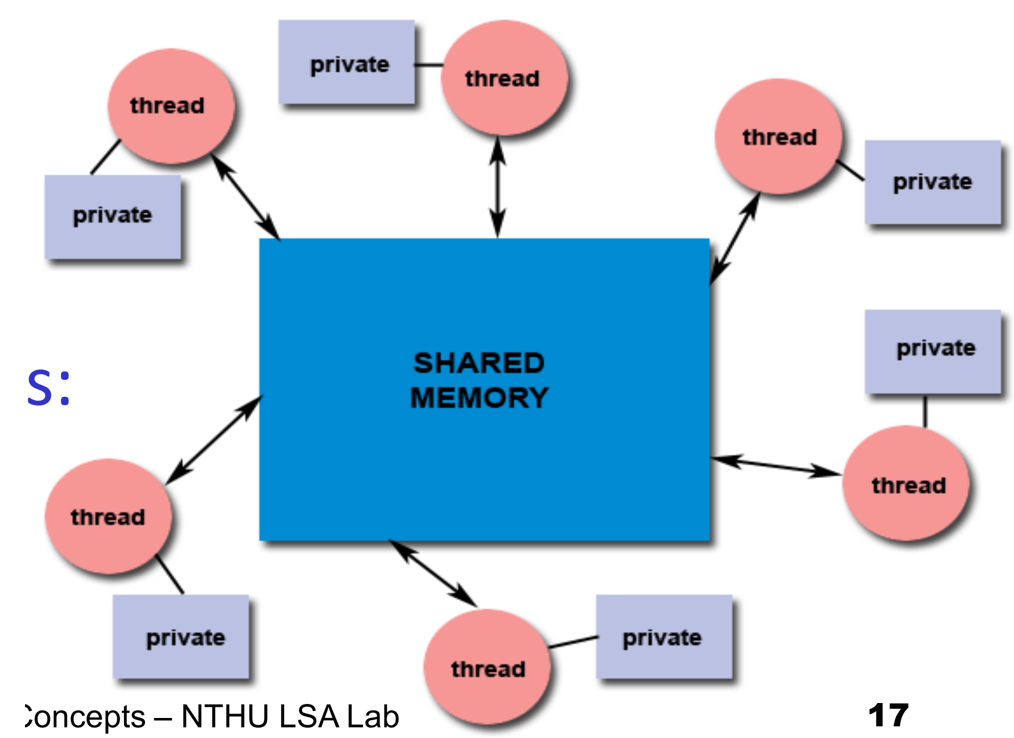
- voluntary ***exit***
  - explicitly: when a thread calls the thread-exit function, or
  - implicitly: thread returns from the function it was asked to run
- ***cancel*** ("kill") a thread
  - ask a thread to stop running, usually more of a suggestion. Thread could decide when to actually finish. possible to force kill (but messy)

Preemptive vs. Cooperative Threads

- Default assumption: preemptive
  - i.e., timer interrupt triggers context switch
  - A "threads manager" at user level gets timer interrupt - in the form of a timer **signal handler**
  - user threads package does not see system calls!
- Easier: cooperative threads
  - additional primitive of "thread-yield" to other threads
  - No preemption: switches context to another thread only by thread-yield or thread-exit
  - However... if a thread does not yield then others can starve!!

Shared-Memory Programming

- Threads communicate through shared memory
  - No need to set up shared memory across processes! Can use globals directly!!
- Issues
  - Synchronization, deadlock, cache coherence
- Programming techniques
  - parallelizing compiler
  - Threads (Pthreads, Java)



Asynchronous vs Synchronous Threading

- Asynchronous threading
  - created thread runs independently and concurrently
  - little dependence, mostly for servers (thread pool) and UI
- Synchronous threading
  - the thread creator waits for created threads to finish and join
  - analogous to fork() parent calling wait() on children

Implementations

- Python threads
- Pthreads (POSIX threads)
- Java threads, Fork-Join library
- OpenMP -
  - compiler directive + API for shared-memory machines

Python3 threads

- `import threading`
  - threads package for all thread use plus synchronization
- `import time`
  - `time.sleep(t)` to yield to another thread
- Issue
  - implementation runs one thread at a time, not in parallel due to global interpreter lock (GIL)
  - To run in parallel, use `multiprocessing` module (processes)

Example Python3 thread: Producer-Consumer

```python
import threading
import time

dataAvail = False
sharedVar = ''

def Producer():
    import string
    global dataAvail
    global sharedVar

    for i in string.ascii_uppercase:
        sharedVar = i
        dataAvail = True
        while dataAvail:
            time.sleep(1)
```

```python
def Consumer():
    global dataAvail
    global sharedVar

    while True:
        while not dataAvail:
            time.sleep(1)
        print(sharedVar)
        dataAvail = False

if __name__ == '__main__':
    p = threading.Thread(target=Producer)
    c = threading.Thread(target=Consumer)
    p.start()
    c.start()
```

Discussion of producer-consumer example in Python3

- easy to write - attach a function to thread
- preemptive threads!
  - will context switch even if they don't sleep
  - try replacing `timer.sleep(1)` with `pass`
- Note explicit `.start()` on created thread
  - contrast to POSIX thread - automatically started when created!
- Should call either .join() or .detach() to free up thread after it finishes
  - but.. here relying on process termination to clean up threads

Alt. option for Python: Generator

- generator
  - function `yield` value instead of `return`
    => continues execution after yield
- Styles: pull vs. push
  - caller **pulls** data by `next(g)` to get data yielded by `g`; or
  - caller **pushes** value by `g.send(value)`, so g receives from `yield` as an expression

Python3 generator (yield) as consumer <u>pulling</u> producer

```
def Producer():
    import string
    for i in string.ascii_uppercase:
        yield i

# consumer as main
if __name__ == '__main__':
    # for-loop instantiates
    # g = Producer() and calls
    # c = next(g) for you until done
    for c in Producer():
        print(c)
```

- for loop instantiates generator and calls next() to pull from generator

- No threads
- rendezvous
- shorter, easy to understand
- lower overhead
- But.. this is mainly for producer-consumer pattern, not thread replacement

Python3 send() to generator as producer pushing to consumer

```
def Consumer():
    while True:
        c = yield # receive
        print(c)
# producer as main loop
if __name__ == '__main__':
    import string
    g = Consumer()
    next(g) # to kickstart consumer
    for i in string.ascii_uppercase:
       g.send(i)
```

- `g.send(val)` pushes to generator instance; `c = yield` receives `val`

- No threads
- rendezvous
- slightly more code than pull, but still simple
- strictly speaking, this is zero-buffer message passing

Thread library

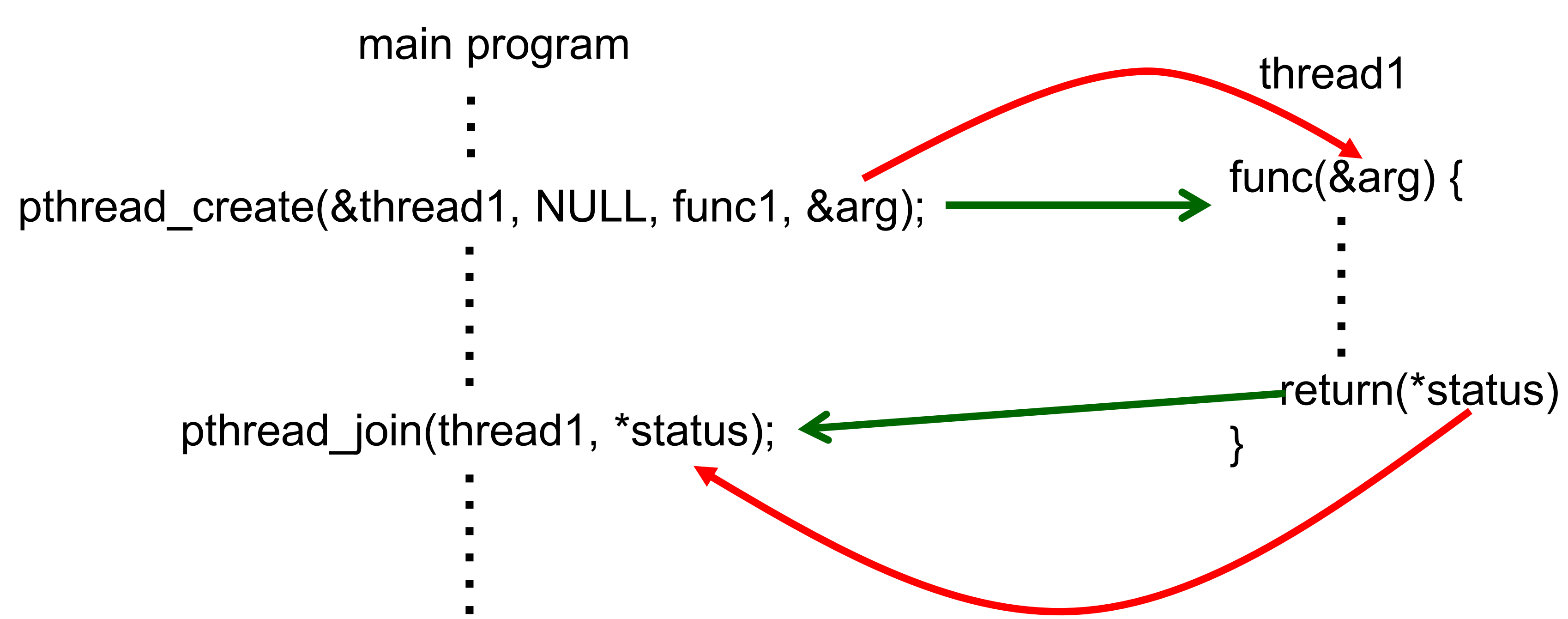
- User space vs kernel space
  - entirely in user space:
    - no system call by user code,
    - though preemptive threads manager needs to call **signal()** to register callback on system events (timer)
  - kernel-level library support
    - make system calls to kernel for thread primitives
- Examples
  - POSIX Pthreads, Java, and Windows

Pthreads

- Pthreads = POSIX threads
- POSIX = Portable Operating System Interface
  - standard for portability across Unix-like systems (Solaris, Linux, Mac)
  - Pthreads = threads implemented to POSIX standard
    IEEE 1003.1c API => this is a spec, not implementation
- Why Pthreads
  - previously, each hardware implements its own proprietary version, not portable
- Similar concept as MPI for message passing libraries

Pthread creation

- pthread_create(thread, attr, routine, arg)
  - thread: a unique id (token) for the new thread
  - attr: thread attribute to set; NULL for default value
  - routine: the function to run after thread is created
  - arg: a single argument to pass to the routine



Example

```c
#include <pthread.h>
#include <stdio.h>
long threadParam[] = { 1, 8, 19, 23, 37 };
#define NUM_THREADS (sizeof(threadParam) / sizeof(long))
void *PrintHello(void *threadId) {
    long *data = (long*)threadId;
    printf("Hello world! I am #%ld\n", *data);
    pthread_exit(NULL);
}

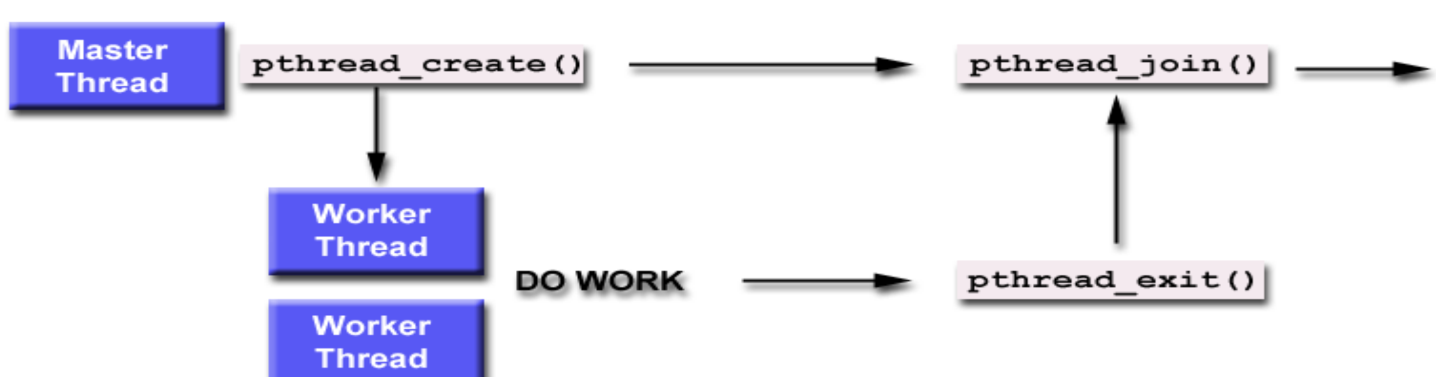
int main(int argc, char*argv[]) {
    pthread_t threads[NUM_THREADS];
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_create(&threads[i], NULL, PrintHello, threadParam + i);
    }
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i]; NULL);
    }
}
```

creates a new thread that runs PrintHello(threadParam[i])

wait for each thread to finish

Pthread join and detach

- `pthread_join(threadId, status)`
  - blocks until specified threadId thread terminates
  - Once way to synchronize between threads
  - Example: a pthread barrier
    ```
    for (int i = 0; i < n; i++) pthread_join(thread[i], NULL);
    ```
- `pthread_detach(threadId)`
  - mark a thread so when it finishes, its resources can be reclaimed (without main thread calling join())
  - once a thread is detached, it can never be joined



Pthread summary

- Portable
  - to different machines, possibly languages too
  - trivial to share data, done in familiar language
- Pitfalls
  - low level code crafting, easy to get race condition
  - not always natural to express code as explicit threads

Thread Pool

- Create a number of threads in a pool
  - standing by, ready to do work
  - recycle back into pool when done
- Pros
  - usually faster to service a request using thread from pool than creating a new thread
  - allows the number of threads in the application to be bound to the pool size
- Bound on #threads
  - could be #CPU cores, #expected requests, memory capacity

Java threads

- Two ways to define threads in Java
  - extending `Thread` class
  - implementing the `Runnable` interface

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

- implemented using a thread library on the host system
  - Win32 threads on Windows
  - Pthreads on Unix-like systems
- Thread mapping depends on JVM implementation
  - Windows 98 or NT: one-to-one   Solaris2: may-to-many

Java Fork-Join Library

- For divide-and-conquer problems
  - fork threads to do concurrent work ("divide"), usually involving recursion
  - Purpose: expose series-parallel parallelism
- Java fork-join library
  - spawn a thread to do the recursive call if the subproblem is sufficiently large; otherwise don't fork a thread.
  - join the recursive calls after complete
  - implementation uses **thread pool** ("ForkJoinPool")

OpenMP

- Motivation:
  - want to write serial program with (few) annotation, let compiler turn into threads
  - portable, though implementation dependent
- Programmer's view
  - requires compiler support: C, C++, Fortran
  - program divided into serial and **parallel regions**
  - compiler parallelizes and make threads, takes care of race conditions

OpenMP pragma

- Syntax
  - `#pragma omp parallel`
  - Create as many threads as there are cores
- parallel for-loop
  - ```
    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
    ```
- Compiler automatically manages synchronization
    ```

OpenMP Restrictions

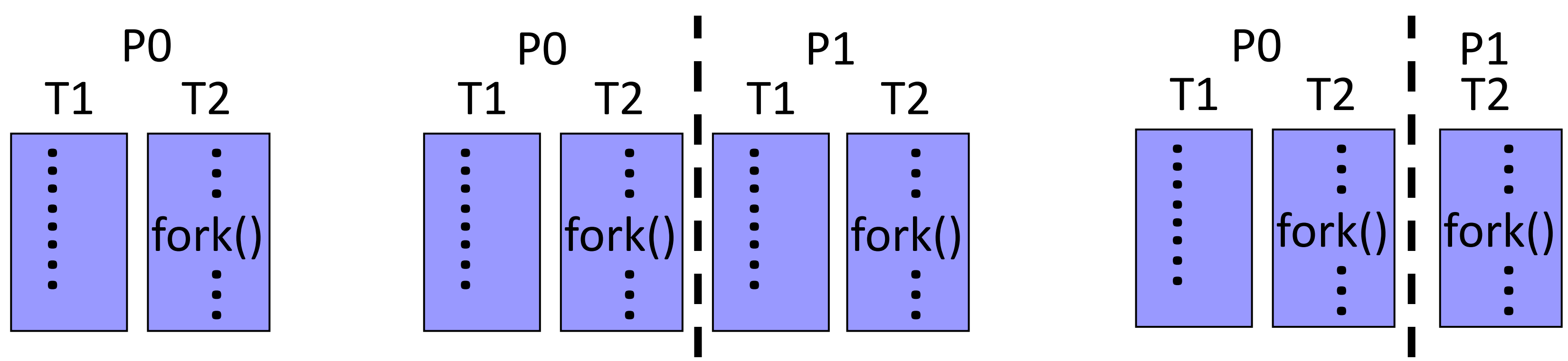
- for-loop restrictions
  - Loop index: signed integer
  - Termination Test: <,<=,>,=> loop invariant int
  - incr/decr by loop invariant int; change each iteration
  - Count up for <,<=; count down for >,>=
  - Basic block body: no control in/out except at top
- Synchronization
  - implicit barrier before and after parallel constructs, but could be removed (nowait)
  - explicit synchronization by critical or atomic

Threading Issues

- semantics of `fork()` and `exec()`
  - duplicate threads or not?
- Signal handling
  - where should a signal be delivered?
- Thread cancellation
  - asynchronous or deferred
- Thread-local Storage
- Scheduler Activations

semantics of `fork()` and `exec()`

- Does `fork()` duplicate threads?
  - POSIX `fork()` duplicates only the thread that calls fork(), but says fork() should be called only from single threaded!
  - Solaris's own (not POSIX) fork API duplicates all threads. Others have two versions of fork()
- `execlp()` only one semantic, not an issue
  - replaces entire process, so no need to duplicate all threads.



Signal Handling

- Signals = callback by OS to user process
  - signal handler is called by OS to handle signals to notify a process that an event has occurred
  - For more information, type `man signal`
- Examples
  - synchronous: illegal memory access
  - asynchronous: user types `Ctrl-C` to kill a process

Signal Programming

- `#include <signal.h>`
  - `SIGALRM` – alarm clock
  - `SIGBUS` – bus error
  - `SIGFPE` – floating point arithmetic exception
  - `SIGINT` – interrupt (i.e., `Ctrl-C`)
  - `SIGQUIT` – quit (i.e., `Ctrl-\`)
  - `SIGTERM` – process terminated
  - `SIGUSR1` and `SIGUSR2` – user defined signals
- Register which signal to handle by calling signal()

```
void my_handler(int s) {
    // this function is called
    // when SIGALRM signal
    // is emitted
    // int s is the signal #,
    // like a user-level ISR
    if (s == SIGLARM) {
        ...
    }
}
void main(void) {
  signal(SIGALRM, my_handler);
  // registers callback
  ...
}
```

Signal Delivery

- Single threaded:
  - system calls the registered callback function
- Several options for Multi-threaded
  - to only the thread that is applicable
  - to every thread in the process
  - to only certain threads in the process
  - assign a specific thread to receive all signals for the process

Thread Cancellation

- What happens if a thread terminates before it completes normally?
  - e.g., user cancels web page loading
- Asynchronous cancellation
  - the target thread is terminated immediately
- Deferred cancellation (default option)
  - target thread periodically checks if it should be terminated
  - gets chance to clean up before termination

Deferred Cancellation

- Thread can enable or disable cancellation
  - Thread cancellation is a requests
  - Actual cancellation depends on thread state
  - If disabled => cancellation remains pending until thread enables it
- Thread gets to decide cancellation point
  - Thread calls `pthread_testcancel()` to set cancellation point
  - Cancellation only occurs when thread reaches cancellation point
  - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through *signals*

Thread-Local Storage

- The "global" data within each thread
  - Not shared with other threads
  - 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

- Kernel provides Lightweight Process (LWP)
  - "virtual processors" = kernel threads
  - kernel makes up call to application about events
- Example: before and after blocking
  - just before blocking: upcall informs user threads scheduler, kernel allocates another virtual processor
  - right after unblocking, another upcall tells user thread scheduler to schedule another virtual processor

Threads on Linux

- Linux as OS does not support multithreading
  - Use various Pthreads implementation (user-level)
- Process creation on Linux
  - `fork()`: creates a new process and a copy of the data from parent
  - `clone()`: creates a "task," which may be process or thread depending on level of sharing.

Use of `clone()` in Linux

- Flags to `clone()` indicate level of sharing
  - No sharing flag set => copy all => make process (clone() == fork() in this case)
  - All sharing flags set => spawns a thread

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.

Windows XP threads

- one-to-one mapping
- each thread contains
  - thread ID
  - register set
  - separate user and kernel stack
  - private data storage
- Primary data structures
  - ETHREADS
  - KTHREADS
  - TEB
- Also provides support for a fiber library => many-to-many