CS 3214: Computer Systems
Lecture 12: Multi-threading

Instructor: Huaicheng Li

October 3 2024

VIRGINIA TECH™

Application-Level Concurrency

- ***The need to pursue multiple, concurrent computations simultaneously within a process besides process-level concurrency …***
- Parallelization: exploit multi-cores for fast parallel task executions
- Multiplexing of I/O and computation
  - CPU is fast, I/O is slow
  - Wasteful for CPUs to wait for I/Os
- Foreground and background activities
  - E.g., VSCode: handle your inputs in the foreground, downloading updates in the background
  - Many other GUI applications
- Handle multiple clients
  - E.g., network server

A New Abstraction - Threads

- *Multiple threads of execution within one process*
- *Each thread has separate logical flows of control*
- *Each thread is part of the hosting process, but with some of its own private context*
  - *Share code, data, kernel context*
  - *Thread's individual stack for local variables (not protected from other threads)*
  - *Each thread has its own thread id (TID)*
- *Think of threads as multiple programs executing concurrently within a shared process, sharing all data and resources, but maintaining separate stacks and execution state.*

	share	do not share
Processes	machine resources, files on disk, inherited file descriptors, terminals	address space
Threads	address space[1], open file descriptors	stack[2] & registers

Quick Recap: Process

- Process = process context + code, data, and stack

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernel context:
- VM structures
- Descriptor table
- brk pointer

Stack
Shared libraries
Run-time heap
Read/write data
Read-only code/data

VM: virtual memory

A *Single-Threaded* Process

Thread (main thread)

Stack

Thread context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Code, data, and kernel context

Shared libraries

Run-time heap

Read/write data

Read-only code/data

Kernel context:
- VM structures
- Descriptor table
- brk pointer

A Multi-Threaded Process

Thread (main thread)

Stack-1

Thread context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Peer Thread

Stack-2

Thread context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Code, data, and kernel context

Shared libraries

Run-time heap

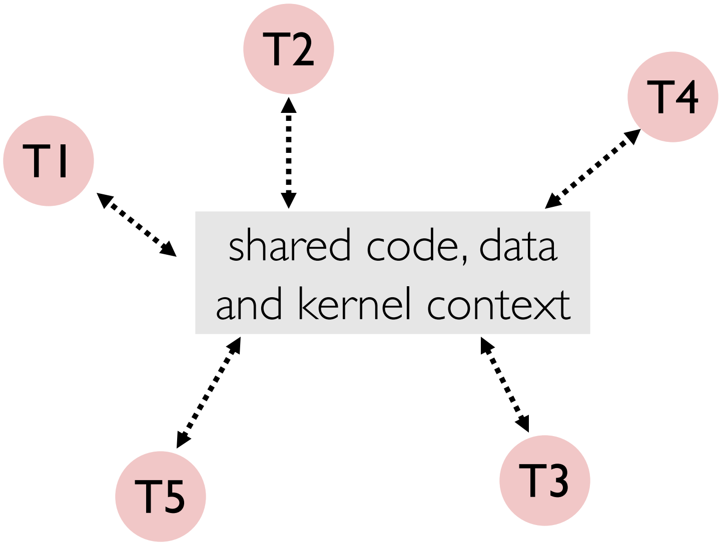
Read/write data

Read-only code/data

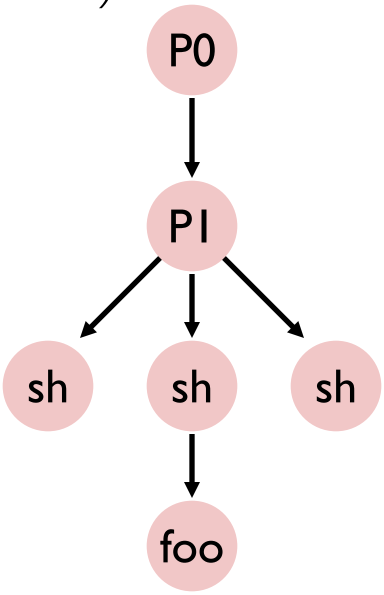
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Logical View of Threads

- Threads are peers to each other
- Processes are organized in hierarchies (parent/child)



Threads



Processes

Thread Scheduling/Concurrency

- Two threads are concurrent if their flows overlap in time
- Otherwise, sequential

Threads vs Processes

- Similarities
  - Each has its own logical control flow
  - Each can run concurrently with others (e.g., on different cores)
  - Each is context switched
- Differences
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are more lightweight than processes
    - Process control (creation/destroy) 2x as expensive as thread control
    - E.g., on Linux
      - ~20K cycles to create and release a process
      - ~10K cycles (or less) to create and reap a thread
    - ...

POSIX Threads (Pthreads)

- Standard interface for thread management, ~60 functions
  - **De facto standard for Unix-like OS, specified in IEEE Std.1003.10-2017**
- Creation and reaping threads
  - *pthread_create()*
  - *pthread_join()*
- Get thread ID
  - *pthread_self()*
- Terminating threads
  - *pthread_cancel()*
  - *pthread_exit()*
- Synchronization primitives on shared variables

```
#include <pthread.h>
int
pthread_create(pthread_t       *thread,
         const pthread_attr_t *attr,
               void           *(*start_routine)(void*),
               void           *arg);
```

Thread Example & Execution

```
void *thread(void *arg);

int main()
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);

    return 0;
}
```

```
void *thread(void *arg)
{
    printf(“Hello, world!\n”);
    return NULL;
}
```

Beyond Pthreads

- Does the ability to maintain multiple flows of control require support from the underlying OS kernel?
- Can it be implemented purely using libraries, etc. using non-privileged instructions and other facilities at user-level?

Cooperative Multi-Threading (User-Level)

- It's possible to maintain multiple control flows entirely *without kernel level support*
- Exists in multiple variants in different languages, known as coroutines or user-level threads depending on variant
- Requires a primitive that saves & restores execution state
- Non-preemptive model: threads' access to the CPU is not preempted (taken away) unless the thread yields access to the CPU voluntarily
- Yield may be directed (saying which coroutine should run next) or undirected (run something else next), e.g. uthreads example
- In some higher-level languages, functions can "yield" temporary results as their execution state is saved and restored (e.g., Python yield)
- Can be combined with asynchronous I/O: yield a promise object that represents an in-progress operation: async/await

Cooperative Multi-Threading

- Pros
  - No OS support required
  - Very lightweight, fast context switch
  - Absence of certain data races, e.g. a++ atomic
  - Scalable when combined with async I/Os
- Cons
  - No multi-core parallelism
  - No explicit preemption (causing starvation)
  - Blocking I/O system calls will block the entire process

Kernel-supported Threads

- **Parallelism** (yes!)
  - using multi-cores/cpus b/c now the OS does the scheduling
  - under I/O, the thread can be moved to BLOCKED state

- **Scheduling** threads like processes, process states

- **Preemption** (yes!)
  - allow shared accessed to a CPU, despite the willingness of multi-threads
    - threads that don't yield can still be preempted
    - the OS can forcefully interrupt the thread and move them to STEADY state

- Kernel-supported threads are the dominant model to use today ...
  - Approaches to implement threads: user-level threads vs. kernel-supported threads
  - kernel threads are a different concept: tasks that run as part of the kernel/OS

Hybrid Models

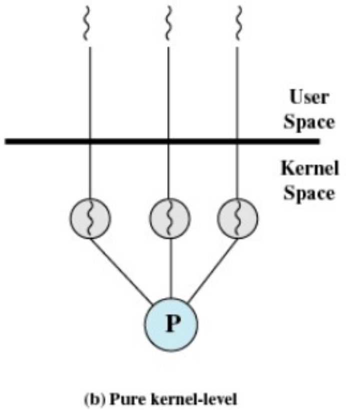


Figure 1: 1:1 model

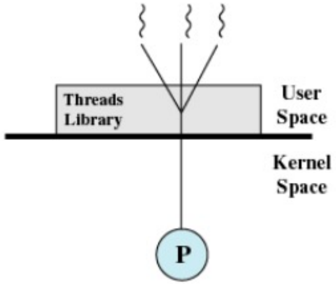


Figure 2: 1:N model

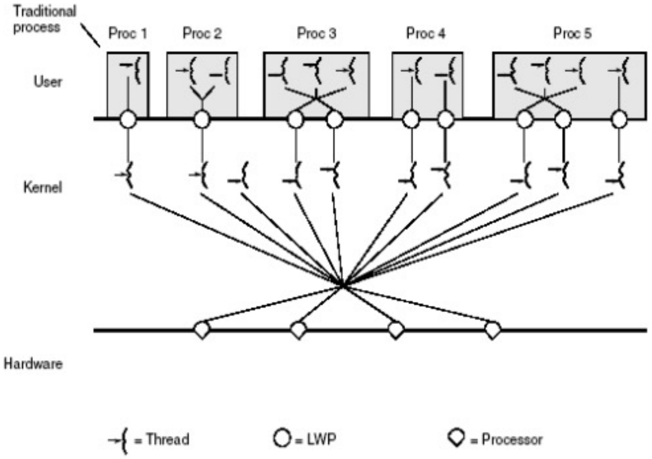


Figure 3: M:N model

LWP: lightweight processes

Hybrid Models (cont)

- Pure user-level threading uses a 1:N model (N user-level threads share 1 OS-level thread)
- Pure kernel-level threading uses a 1:1 model (1 OS thread for each user thread)
- Hybrids (M:N) models try to obtain the best of user-level and kernel-supported threads.
- Examples: Windows Fibers, (now defunct) Solaris M:N model
  Increase in complexity (and lack of payoff) led to the M:N model being largely abandoned.
- Heavy investment/optimization in reducing the costs of the 1:1 model, e.g. fast user-level synchronization facilities

Threads Downsides

- Too easy to share resources (?)
  - Not much control over scheduling
  - Difficult to debug (ordering unpredictable)

Concurrency Management

- Applications rarely create separate, new threads for individual tasks, particularly if small
- Instead, they manage the number of threads needed to perform work and distribute work to threads
- Trade-off:
  - Too many threads: leads to increased contention for resources and resulting overhead from managing that
  - Too few threads: risks underutilization of CPUs/cores
- **Target:** number of READY + RUNNING threads around equal to number of cores
- **Solution:** thread pools

Pseudocode Source: Lea [1]

```
Result solve(Param problem) {
  if (problem.size <= GRANULARITY_THRESHOLD) {
    return directlySolve(problem);
  } else {
    in-parallel {
      Result l = solve(lefthalf(problem));
      Result r = solve(rightHalf(problem);
    }
    return combine(l, r);
  }
}
```

Challenge

An execution framework must map the tasks created in `in-parallel` to threads.

Concurrency under Threads

```c
static volatile int counter = 0;

// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n",
            counter);
    return 0;
}
```

Concurrency Primitives (Next Few Lectures)

- Semaphore
- Mutex
- Lock
- Conditional Variables