System Calls

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

Topics

- processes vs. threads
- cons/pros of threads
- thread pool
- POSIX threads

Threads

Threads TL;DR

- threads are similar to processes
- but there are some important differences

- TL;DR
  - threads are **more efficient** than processes
  - threads are **more difficult** to use correctly

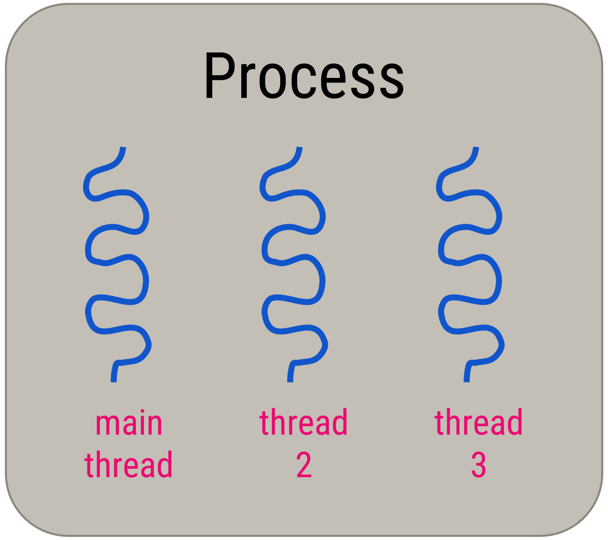
Threads

- just like processes, threads can also be used to express parallelism
- if we need multiple tasks to run concurrently, we can:
  - run each task in separate process; or
  - run each task in separate thread
- if we have enough CPUs, each task can run on separate CPU
- the most common use of threads is to **speed up execution** by allowing programs to utilize multiple CPUs/cores
- programs that use multiple threads are called multi-threaded
- programs that don't use multiple threads are called single-threaded

Threads

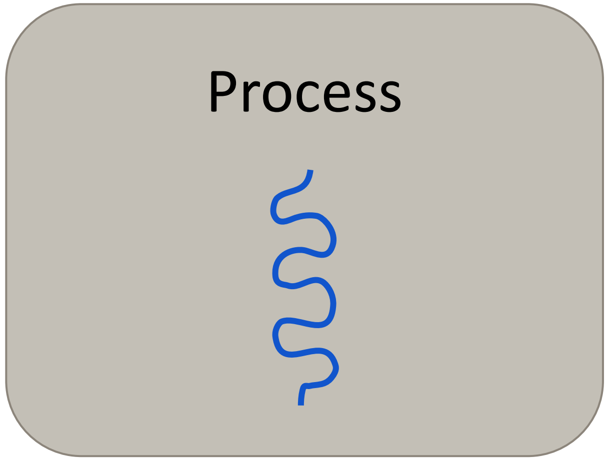
- every process starts with one thread, but can add more threads
  - the original thread is usually called the main thread
  - main thread is the one executing `main()`
- a thread cannot exist without a process
- a process acts like a container for all its threads
- all threads within one process **share the resources** of the process
- threads are scheduled and execute independently

- analogies:
  - multiple VMs share resources of the host computer
  - multiple processes share the resources of the OS



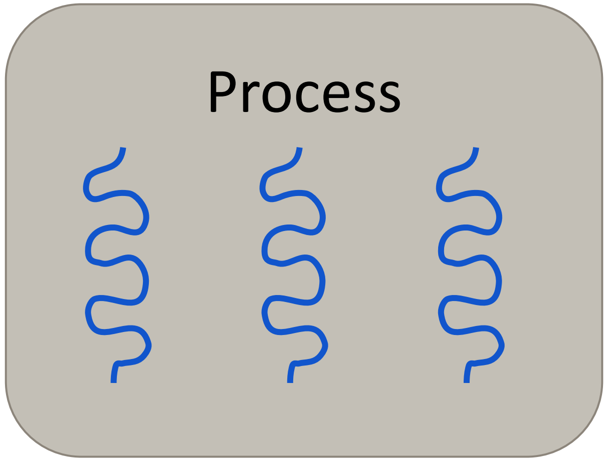
Process with 1 thread (single-threaded process)

- think of a process as a way to group related resources together
  - e.g. address space (heap, global variables, etc), open files, sockets, child processes, signal handlers, accounting info
- a process also has aa "thread of execution"
  - consisting of registers, stack and state
- every process starts with a single thread of execution

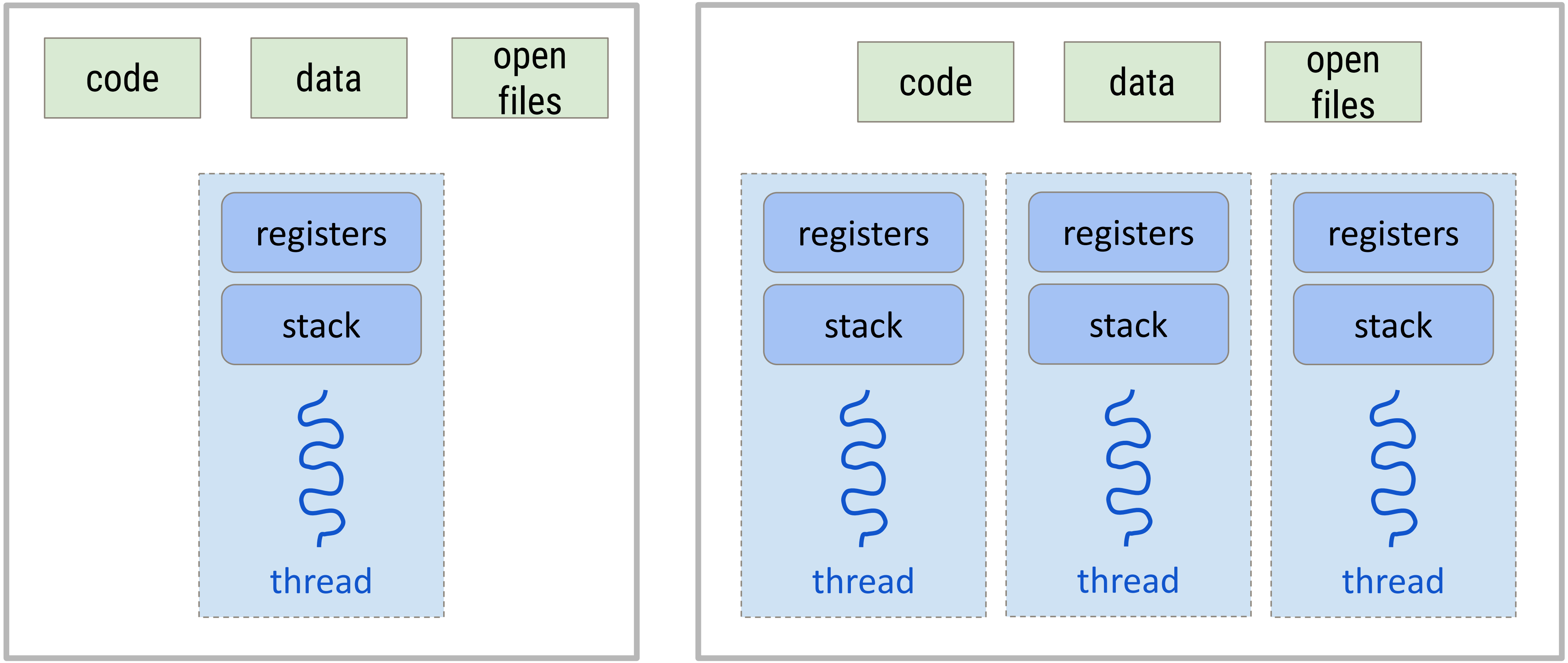


Process with many threads

- any process can create additional thread(s)
- threads allow **multiple executions** to take place within one process environment
- think of threads as multitasking within one process
- all threads execute simultaneously, and are scheduled independently
- threads can make system calls simultaneously
- a thread can share many/most resources with other threads in the same process
- threads belonging to different processes do not share anything



Single-threaded v.s. multi-threaded processes



single-threaded process

multi-threaded process

Process and thread items

Per-process items shared by threads	Per-thread items not shared by threads
address space	PC
global variables	registers
heap	stack
open files	state
child processes	local variables
accounting information	...
signals	
...	

For example...

- if one thread opens a file, all* threads can read and write to it (but very carefully)
- if one thread changes a global variable, the change will be visible in all* other threads
- if one thread calls `exit()`, all* threads will be killed (*all thread in the same process)

Why Threads

Why threads?

- multithreaded applications could run faster on computers with multiple CPUs/cores
  - by dividing work into tasks and then running tasks in separate threads
  - with N cpus/cores, the optimum speedup is N
- threads can be used to parallelize I/O
  - e.g. 2 threads, each reading a different file
- threads can be used to write responsive GUI applications
  - one UI thread + many worker threads executing lengthy operations, such as I/O requests
  - example: browser running Discord in one tab and YouTube video in another
- using multiple threads can sometimes lead to simpler design
  - e.g. threads can be used to avoid using non-blocking, asynchronous I/O with callbacks and/or complicated state machines

Why threads?

- compared to processes, threads …
  - are "lighter weight"
  - use less memory
  - usually faster to create and destroy
  - have more* options for communication via shared memory
  - can be context-switched more efficiently

Why not threads?

- if a thread misbehaves or crashes, the whole process could misbehave or crash
- programming with threads is more difficult than with processes, because we have to worry about things like:
  - race conditions
  - deadlocks
  - starvation
- to deal with the above, we need to learn:
  - synchronization mechanisms (e.g. mutexes, spinlocks, barriers)
  - atomic operations
  - deadlock avoidance techniques

Thread Example

Thread example: static web server

- web server accepts page requests from browsers and sends replies (pages) back

- handling of each request could be broken down into 3 tasks:
  - receiving request
  - locating and reading the corresponding file on disk
  - sending the page back to browser

- how can we write a server that can handle as many requests per second as possible?
  - buy faster hardware
  - use non-blocking system calls
  - use threads

Thread example: static web server

- using threads to speed up the web server

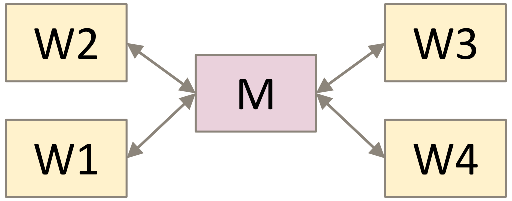
- option 1 – small improvement
  - we treat the tasks as a parallel pipeline with 3 stages
  - we create 3 threads: one for receiving requests, one for fetching results, one for sending pages
  - up to 3x speedup, providing all 3 stages take same amount of time
  - not enough speedup for modern hardware, e.g. 16 core CPU, few SSD disks

- option 2 – much better improvement
  - create separate thread for each request
  - each thread completes 1 request from start to finish (receive, fetch, send)
  - can you guess what the issues might be with this approach?

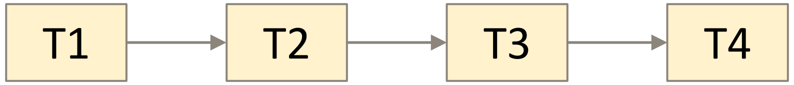
Thread Communication

Common thread communication scenarios

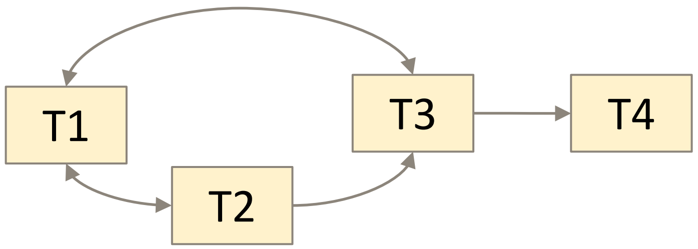
- **manager/worker (**aka master/slave)
  - one manager thread assigns work to worker threads
  - typically manager thread handles all I/O
  - number of worker threads can be static or dynamic



- **pipeline**
  - a task is broken into a series of stages,
    where output of stage (i) is input to to stage (i+1)
  - each stage handled by a different thread



- **other**
  - there a many other more sophisticated ways of organizing threads
  - eg. thread pool, producer/consumer



Thread Pools

Thread pool

- recall the web-server example
  - when server receives a request, it creates a separate thread to handle the request
  - once request is handled, thread is destroyed

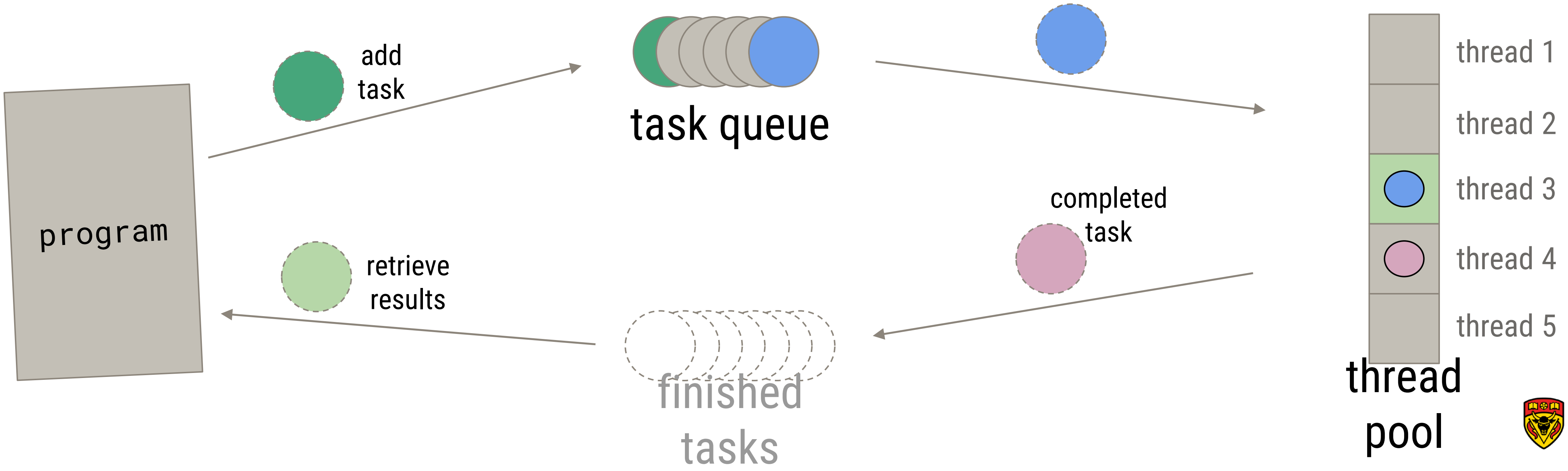
- issues:
  - frequent thread creation and termination → performance problem
  - potentially large number of concurrent threads → resource problem

Thread pool

- **thread pool** — a software design pattern allowing thread recycling/re-use
- main thread creates and maintains a pool of worker threads
- pool size can be tuned, e.g. to the available computing resources, number of cores, ...
- when program needs a thread, it is borrows one of the worker threads from the pool
- when worker thread is done, program returns it back to the pool
- benefits:
  - thread creation/destruction costs are minimized
  - maximum number of concurrent threads is limited
- problems:
  - what if the program needs more threads than the size of the pool?

Thread pool + task queue

- thread pools are usually combined with task queues
- when a program needs to execute task in parallel, instead of asking for a thread, it inserts the task into a task queue
- thread pool monitors the task queue, and next available thread takes task from the task queue, and finishes it
- task queues could implement advanced features, such as priorities and dependencies



Thread Libraries

Thread libraries

- a thread library provides the programmer with an API for creating and managing threads
- a thread library typically contains higher level wrappers around low level system calls
- examples
  - POSIX threads, a.k.a. pthreads (mostly for UNIX)
  - C++ threads (portable)
  - Win32
  - Java

POSIX threads (pthreads)

- to use POSIX threads
  - □ `#include <pthread.h>`
  - compile with `-pthread` (on older g++ compilers use `-lpthread`)
- `pthread_create(*threadid, attr, start_routine, arg);`
  - starts a thread and calls `start_routine(arg)` in new thread; similar to `fork()`
  - each thread gets unique `threadid`, which we need to keep
- `pthread_exit(status);`
  - terminates the current thread, similar to `exit()`, or you can `return` from `start_routine`
- `pthread_join(threadid, *status);`
  - blocks the calling thread until the specified thread terminates, similar to `wait()`
- `pthread_attr_init(attr)` and `pthread_attr_destroy(attr);`
  - initializes / destroys thread attributes
  - these can be fine-tuned with `pthread_attr_set_?()` functions

Example: multithreaded "Hello world"

```
#include <pthread.h>
#include <stdio.h>
#include "slow_printf.h"

void * task(void *) {
  /* this runs in new thread */
  slow_printf("Hello\n");
  pthread_exit(0); // or return 0;
}

int main() {
  pthread_t tid;
  pthread_create(&tid, NULL, task, NULL);
  /* this runs in original (main) thread */
  slow_printf("world\n");
  pthread_join(tid, NULL);
  printf("Done\n");
}
```

Compile with:

```
$ gcc -pthread main.c
```

Possible outputs:

```
Hello
world
Done
```

```
world
Hello
Done
```

```
wHoerllldo

Done
```

https://replit.com/@jonathanwhudson/hello-world-1

Shared Variables

Multithreading & shared (global) variables

- since address space is shared between threads, all global variables are shared by default
- if one thread changes a global variable, it changes for all threads
- this is very different behavior from multi-process programs (where we used `fork`)

Processes & global variables

```c
int x; /* global variable */

void do_something() {
    x = 11;
    exit(0);
}

int main() {
    x = 10;
    int pid = fork();
    if( pid == 0) {
        do_something();
    }
    else {
        while( wait(NULL) != -1);
    }
    printf("x=%d\n", x);
}
```

Output:

```
$ ./a.out
???
```


UNIVERSITY OF CALGARY

Processes & global variables

```
int x; /* global variable */

void do_something() {
    x = 11;
    exit(0);
}

int main() {
    x = 10;
    int pid = fork();
    if( pid == 0) {
        do_something();
    }
    else {
        while( wait(NULL) != -1);
    }
    printf("x=%d\n", x);
}
```

Output:

```
$ ./a.out
x = 10
```

https://repl.it/Lulm/1

UNIVERSITY OF CALGARY

Threads & global variables

```c
int x; /* global variables are shared between threads
!!! */

void * do_something(void *) {
    x = 11;
    pthread_exit(0); // or return 0;
}

int main() {
    x = 10;
    pthread_t tid;
    pthread_create( & tid, NULL, do_something, NULL);
    pthread_join( tid, NULL);
    printf("x=%d\n", x);
}
```

```
Output:

$ gcc -pthread
thread.c
$ ./a.out
???
```

Threads & global variables

```
int x; /* global variables are shared between threads
!!! */

void * do_something(void *) {
    x = 11;
    pthread_exit(0); // or return 0;
}

int main() {
    x = 10;
    pthread_t tid;
    pthread_create( & tid, NULL, do_something, NULL);
    pthread_join( tid, NULL);
    printf("x=%d\n", x);
}
```

Output:

```
$ gcc -pthread
thread.c
$ ./a.out
x = 11
```

https://repl.it/LuoF/0

Creating Multiple Threads

Creating variable number of threads

- how do we create multiple threads?
- we need to keep track of all thread IDs that we create, so that we can join the threads later
- we'll need an array to store these

Example with multiple threads

```
#include <pthread.h> #include <stdio.h> #include <stdlib.h>

#define NUMBER_OF_THREADS 5

void * thread_print(void * tid) {
  printf("thread %ld running\n", (long int) tid);
  pthread_exit(0);
}

int main() {
  pthread_t threads[NUMBER_OF_THREADS];
  for ( long i = 0; i < NUMBER_OF_THREADS; i++) {
    printf("creating thread %ld\n", (long int) i);
    long status = pthread_create(&threads[i], NULL, thread_print, (void *) i);
    if (status != 0) {
      printf("Oops, pthread_create returned error code %ld\n", status);
      exit(-1);
    }
  }
  for (i = 0; i < NUMBER_OF_THREADS; i++)
    pthread_join(threads[i], NULL);
  return 0;
}
```

Compile with:

```
$ gcc -pthread thread.c
```

Can you guess the output?

```
#include <pthread.h> #include <stdio.h> #include <stdlib.h>

#define NUMBER_OF_THREADS 5

void * thread_print(void * tid) {
  printf("thread %ld running\n", (long int) tid);
  pthread_exit(0);
}

int main() {
  pthread_t threads[NUMBER_OF_THREADS];
  for ( long i = 0; i < NUMBER_OF_THREADS; i++) {
    printf("creating thread %ld\n", (long int) i);
    long status = pthread_create(&threads[i], NULL, thread_print, (void *
    if (status != 0) {
      printf("Oops, pthread_create returned error code %ld\n", status);
      exit(-1);
    }
  }
  for (i = 0; i < NUMBER_OF_THREADS; i++)
    pthread_join(threads[i], NULL);
  return 0;
}
```

Possible output:

```
$ ./a.out
creating thread 0
creating thread 1
thread 0 running
creating thread 2
creating thread 3
thread 2 running
thread 1 running
creating thread 4
thread 3 running
thread 4 running
```

Other possible outputs:

https://repl.it/Luid/0

Passing multiple parameters to threads & retrieving results

- pthread interface only allows a single parameter to be passed to the thread function
- lucky for us, the parameter is a void pointer (`void*`), a generic pointer
- we can use it to pass any number of parameters, and even use it to return results
- a common design pattern is to create an array of struct, one for each thread
- let's say we want N threads to compute `result = a + b * c` for different values of `a`, `b` and `c`

```
#define N 5
struct TMem {
  int a, b, c;   // inputs
  int result;    // outputs
  pthread_t tid;
} tarr[N];
```

- then we can pass a pointer to different elements of this array to each thread
- basically, each thread will get its own dedicated area of memory

allocate separate memory for each thread, including input parameters and result

```c
#define NUMBER_OF_THREADS 5
struct TMem {
  pthread_t tid;
  int a, b, c, result;
} tarr[NUMBER_OF_THREADS];

void * calc(void * targ) {
  struct TMem * tm = (struct TMem *) targ;
  tm-> result = tm->a + tm->b * tm->c;
  return 0;
}

int main() {
  for (int i = 0; i < NUMBER_OF_THREADS; i++) {
    tarr[i].a = i; tarr[i].b = i + 1; tarr[i].c = i + 2;
    if( 0 != pthread_create(& tarr[i].tid, 0, calc, & tarr[i])) {
      printf("Error: pthread_create failed\n"); exit(-1);
    }
  }
  for (int i = 0; i < NUMBER_OF_THREADS; i++) {
    pthread_join(tarr[i].tid, 0);
    printf("%d + %d * %d = %d\n",
      tarr[i].a, tarr[i].b, tarr[i].c, tarr[i].result);
  }
}
```

```
$ gcc -l pthread thread.c
$ ./a.out
0 + 1 * 2 = 2
1 + 2 * 3 = 7
2 + 3 * 4 = 14
3 + 4 * 5 = 23
4 + 5 * 6 = 34
```

https://repl.it/@jonathanwhudson/simple-threads-1

C++ Threads

C++ threads

```
#include <iostream>
#include <thread>

int x = 10;

void do_something() {
  x = 11;
}

int main() {
  auto t1 = std::thread( do_something );
  t1.join();
  std::cout << "x = " << x << "\n";
}
```

```
$ g++ -pthread thread.cpp
$ ./a.out
x = 11
```

https://repl.it/@jonathanwhudson/global-variable

UNIVERSITY OF CALGARY

C++ threads – passing parameters by value

```
#include <cstdio>
#include <thread>
#include <chrono>
#include <string>

void task(std::string task_name, int start, int end) {
  for( int i = start ; i < end ; i ++ ) {
    printf("Thread '%s': i=%d\n", task_name.c_str(), i);
    std::this_thread::sleep_for(
      std::chrono::milliseconds(1));
  }
}

int main() {
  auto t1 = std::thread( task, "t1", 0, 3);
  auto t2 = std::thread( task, "thread 2", 100, 105);
  t1.join();
  t2.join();
}
```

```
$ g++ -pthread thread.cpp
$ ./a.out
Thread 't1': i=0
Thread 'thread 2': i=100
Thread 't1': i=1
Thread 'thread 2': i=101
Thread 't1': i=2
Thread 'thread 2': i=102
Thread 'thread 2': i=103
Thread 'thread 2': i=104
```

https://repl.it/@jonathanwhudson/c-threads-with-parameters

C++ threads – parameters (by reference) & retrieving results

```
void sum( int start, int end, int step, int & result)
{
  for( auto i = start ; i < end ; i += step)
    result += i;
}

int main()
{
  constexpr int N = 1024;
  int sum_even = 0, sum_odd = 0;
  std::thread t1(sum, 0, N, 2, std::ref(sum_even));
  std::thread t2(sum, 1, N, 2, std::ref(sum_odd));
  t1.join(); t2.join();
  std::cout << "Sums = " << sum_even << " " << sum_odd << "\n"
            << "Sum = " << sum_even + sum_odd << "\n"
            << "Formula = " << N * (N-1) / 2 << "\n";
}
```

```
Sums = 261632 262144
Sum = 523776
Formula = 523776
```

https://repl.it/@jonathanwhudson/thread-sum

C++ threads – parameters by pointer & retrieving results

```
void sum( int start, int end, int step, int * result)
{
  for( auto i = start ; i < end ; i += step)
    * result += i;
}

int main()
{
  const int N = 1024;
  int sum_even = 0, sum_odd = 0;
  std::thread t1(sum, 0, N, 2, & sum_even);
  std::thread t2(sum, 1, N, 2, & sum_odd);
  t1.join(); t2.join();
  std::cout << "Sums = " << sum_even << " " << sum_odd << "\n"
            << "Sum = " << sum_even + sum_odd << "\n"
            << "Formula = " << N * (N-1) / 2 << "\n";
}
```

```
Sums = 261632 262144
Sum = 523776
Formula = 523776
```

https://repl.it/@jonathanwhudson/thread-sum-2

C++ threads – lambdas [advanced]

```
int main()
{
  const int N = 1024;
  int sum_even = 0, sum_odd = 0;
  std::thread t1( [&] () {
    for( auto i=0 ; i<N ; i+=2)
      sum_even += i;
  });
  std::thread t2( [& sum_odd] () {
    for( auto i=1 ; i<N ; i+=2)
      sum_odd += i;
  });
  t1.join(); t2.join();
  std::cout << "Sums = " << sum_even << " " << sum_odd << "\n"
            << "Sum = " << sum_even + sum_odd << "\n"
            << "Formula = " << N * (N-1) / 2 << "\n";
}
```

```
Sums = 261632 262144
Sum = 523776
Formula = 523776
```

https://repl.it/@jonathanwhudson/thread-sum-with-lambdas

C++ threads – array of threads

```
const int NTHREADS = 5;

void task(int tid)
{
  printf("thread %d running\n", tid);
}

int main()
{
  std::vector<std::thread> threads;
  for( auto i = 0 ; i < NTHREADS ; i ++) {
    printf("creating thread %d\n", i);
    threads.push_back( std::thread(task, i));
  }
  for( auto & t : threads)
    t.join();
}
```

```
creating thread 0
creating thread 1
thread 0 running
creating thread 2
creating thread 3
creating thread 4
thread 4 running
thread 3 running
thread 2 running
thread 1 running
```

https://repl.it/@jonathanwhudson/array-of-threads

Thread Implementations

Thread implementations

- kernel-level threads
  - managed by the kernel/OS
  - most common

- user-level threads
  - entirely implemented in user space
  - kernel knows nothing about threads (i.e. OS does not need to support threads at all)
  - not very common, used in some HPC environments for efficiency

- hybrids
  - very uncommon (HPC?)

Signal Handling

Signal handling

- signal handling is more complicated with threads
  - which thread should handle the signal?
    i.e. in which thread's context should the signal handler be executed?
  - what about user-level threads?
- in POSIX systems, signal delivery depends on the type of the signal:
  - some signals are thread specific:
    - eg. `SIGSEGV` is delivered to the thread that caused the exception
    - `pthread_kill(thread_id, signal)` is only delivered to the target thread
  - most signals are delivered to the process
    - only one thread will handle the signal (usually the main thread, but can be arbitrary)
    - can change which thread handles which signal using `pthread_sigmask()`
- example:
  - default behavior of <ctrl-c> → `SIGINT`, kills all threads

Thread Example

Thread example: word processor

- you are editing a document with 1000 pages
- on page 1 you delete a paragraph, then you decide to jump to page 900
- the application will be busy re-formatting the entire document from the first page so that the content on page 900 can be displayed correctly

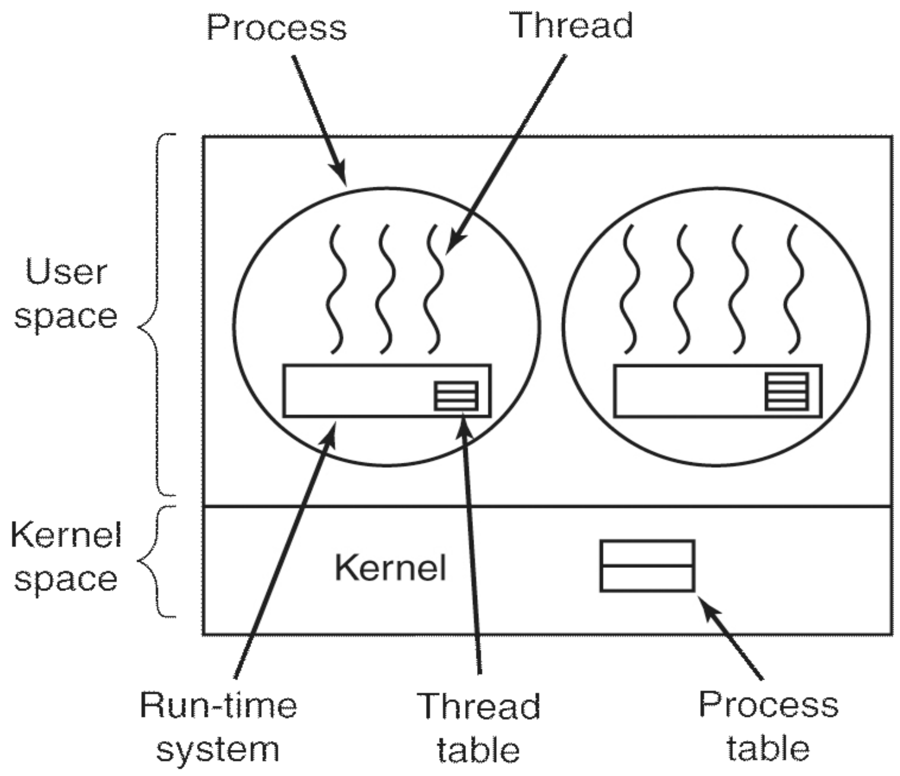
How can threads help?

- one thread for interacting with the user
- one or more threads used for reformatting (to make it run faster on multi-core CPUs)
- one thread for spell checking
- one thread for auto-saving
- ...

User/Kernel Level Threads

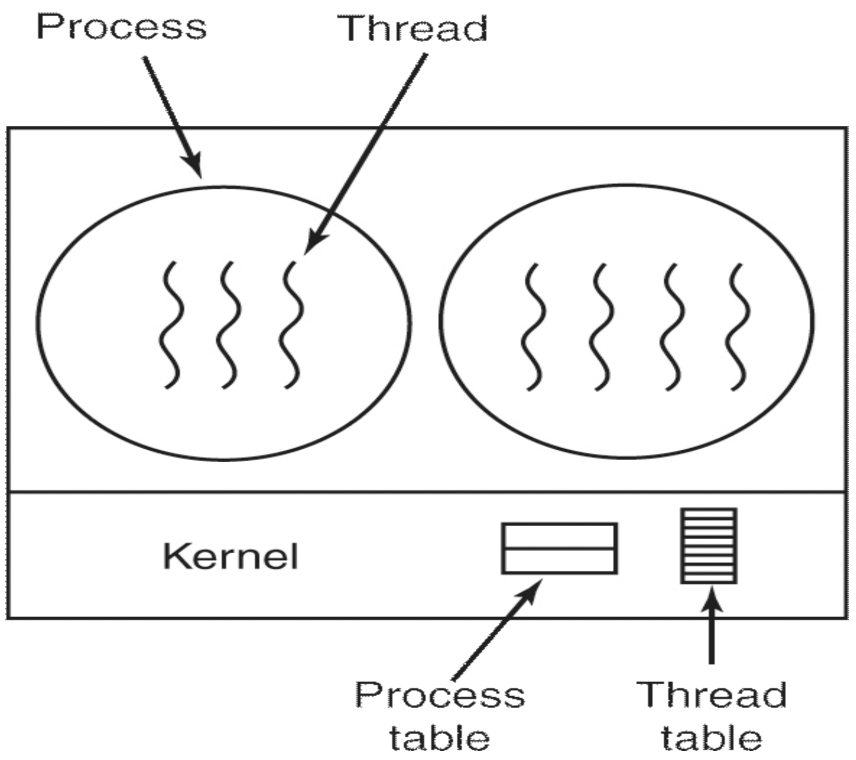
User-level threads

- threads are implemented entirely in user space
- requires no support from OS → can be used on OSes that don't support threads
- each process has its own thread table and scheduler
- threads usually switch only on I/O requests
- no need to trap into kernel when switching threads, so they are very efficient
- allows custom management and scheduling
- requires OS to support non-blocking I/O
- each additional thread makes other threads run slower
- some issues with paging



Kernel-level threads

- one master thread table at the kernel level
- thread creation/deletion/scheduling done in the kernel space
- works well when lot of blocking I/O ops needed
- processes with multiple threads run faster
  - each thread can get same CPU time
- less efficient, since thread operations need to trap into the kernel
- increased kernel complexity

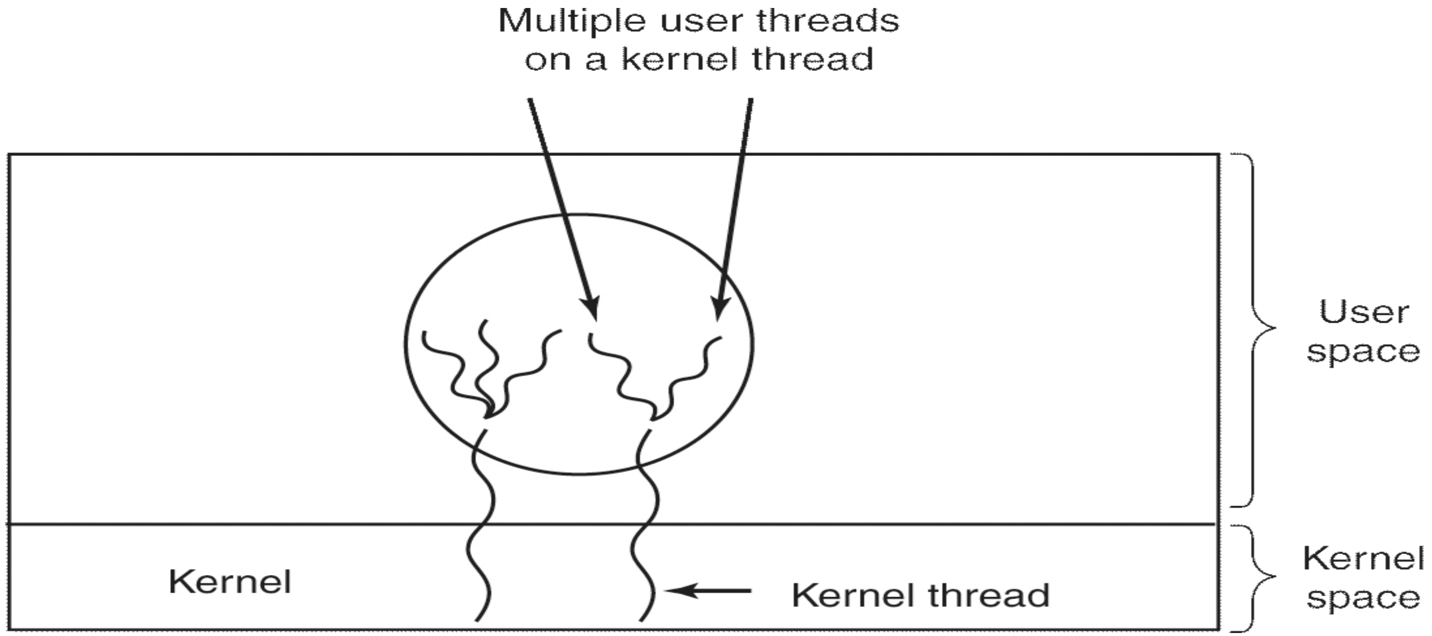


User-level vs kernel-level threads

	Pros	Cons
User level	• no need for OS support • fast context switch • no traps are needed • customized scheduling	• needs non-blocking system calls • a thread may run forever • page faults • inefficient for threads with many blocking procedure/system calls • all threads get one time slice
Kernel level	• blocking calls are no problem • OS aware of all threads → more efficient global scheduling	• some issues around fork() • sending signals to threads

Hybrid

- goal: combining the advantages of user-level threads with kernel-level threads.
- idea: multiplex user-level threads into some or all of the kernel-level threads
  - the kernel is aware of only the kernel-level threads and schedules those
  - the user-level threads are managed in the user space
- it is up to the application to decide how many kernel-level and user-level threads to create
- result: more flexibility



Scheduler Activations

Scheduler activations

- a mechanism to allow closer integration between user-level threads and the kernel
- allows for hybrid kernel-level and user-level threads
- supported by some kernels
- kernel notifies the application when 'interesting' events occur
  - eg. when a thread has been blocked, could deal with page faults
  - the notification is called an **upcall**
  - application can then react by rescheduling its threads

Thread Models

Thread models

- N:1 (many-to-one) or user-level threads
  - many user-level threads per single kernel thread
  - thread management is done by the thread library in the user space
  - E.g., Solaris Green Threads, GNU Portable Threads

- 1:1 (one-to-one) or kernel-level threads
  - maps each user thread to a kernel thread
  - E.g., Windows NT/XP/2000, Linux, Solaris 9 and later

- M:N (many-to-many) or hybrid user/kernel level threads
  - multiplexes many user-level threads to a smaller or equal number of kernel threads
  - eg. Marcel, a multithreading library for HPC

Review

Review

- When the parent process terminates, what happens to its children (UNIX)?
  - https://repl.it/@pavolfederl/GraveExpensiveField
  - but try the same program on your Linux machine
- What could cause a process to change from running state to ready state?
- Why is thread creation faster than process creation?
- What are some of the items that are shared among threads?
- When running multiple threads on a multi-core machine, will all cores be utilized?
- What is the difference between using `pthread_exit()` and `exit()` in a thread?
- Name some pros and cons of implementing threads in user space.

Simple exercise

- write a program that calculates the sum of numbers 1..N
- N will be given on command line
- create 2 threads
  - thread 1:
    - calculates sum of numbers [1 .. N/2)
    - stores result in one global variable
  - thread 2:
    - calculates sum of even numbers [N/2 .. N]
    - stores result in another global variable
- main thread
  - parses command line argument "N"
  - sets 2 global variables to "0" and starts 2 threads
  - waits for both threads to finish
  - sums the two global variables & prints out the result

Summary

- processes vs. threads
- cons/pros of threads
- thread pool
- POSIX threads

Onward to … thread cancellation and race conditions

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY