CPSC 457

Threads

Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

- processes v.s. threads
- cons/pros of threads
- thread pool
- POSIX threads
- thread implementations

Threads

- in many ways threads are similar to processes
  - both can be used to write applications that need some parallelism
- main differences:
  - threads are more efficient that processes
  - threads are more complicated to program correctly

Thread

- informally, a thread is a "process within a process", or "mini process"
- a process can have one or more threads, and a thread is always associated with a process
- all threads within one process share the resources of the process
  - you can think of a process as a container for all its threads
  - a thread cannot exist without a process
- all threads are scheduled independently

- analogies:
  - multiple VMs on a single physical computer share resources of the computer
  - multiple processes in an OS share the resources of the OS

Process vs Thread

Both processes and threads can be used to write concurrent applications,

but there are important differences:

- processes are independent and self contained
- threads exist as "subsets" of a process
- threads belonging to the same process share many/most resources with each other
    - eg. address space, open files
- processes interact only through OS mechanisms (IPC = interprocess communication)
- threads have more options for communication
- processes have easier access to built-in OS mechanisms,

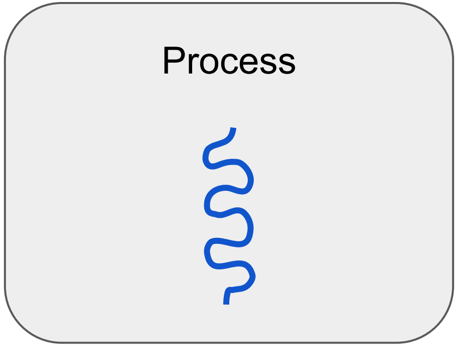
  but they are usually less efficient than threads

Process

- think of a process as a way to group related resources together, eg:

  address space containing program text and data, open files,

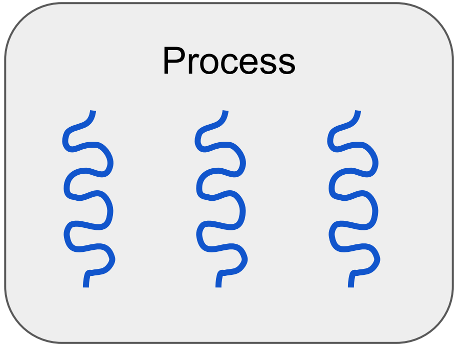
  child processes, pending alarms, signal handlers, accounting info, etc ...
- a process also has a "thread of execution"
  - consisting of registers and a stack
  - by default a process starts with a single thread of execution



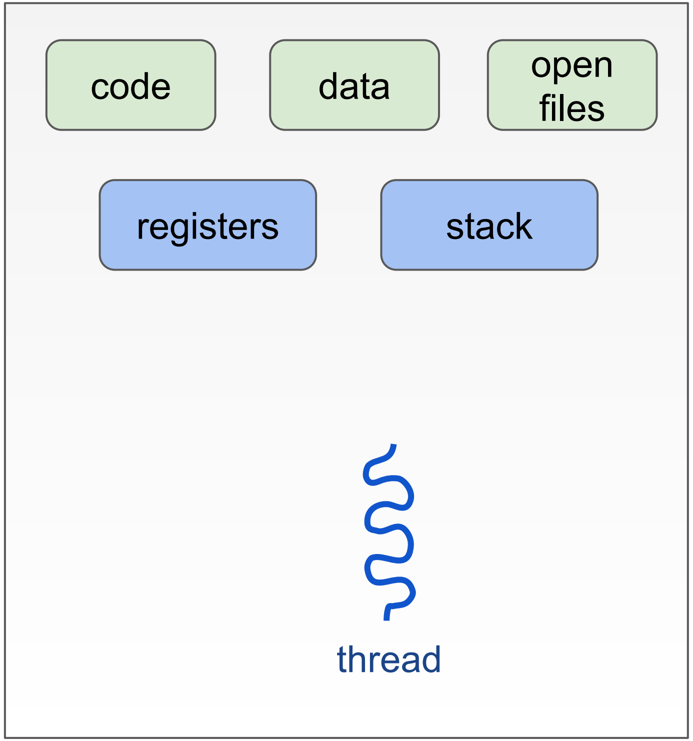
Threads

- threads allow **multiple executions** to take place **within one process** environment
- you can think of a thread as a *mini-processes* within a process,

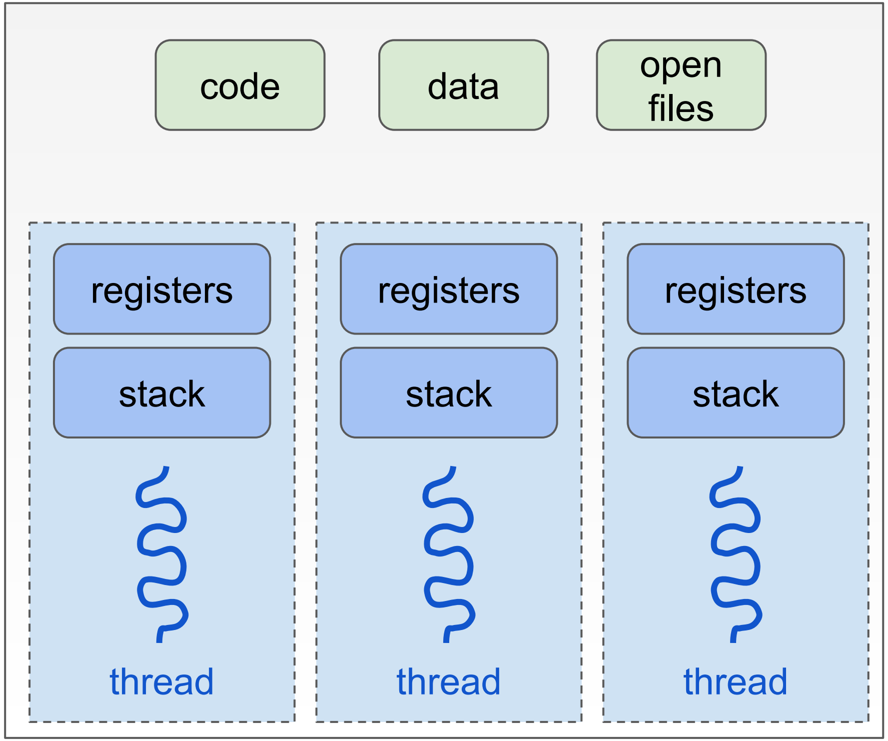
  or a unit of execution of a process
- threads execute simultaneously with other threads in the process
  - can be scheduled independently
  - can make system calls simultaneously
- a thread can share many/most resources with other threads in the same process



Single-threaded v.s. multi-threaded processes



single-threaded process



multi-threaded process

Process and thread items

Per-process items	Per-thread items
address space	registers
global variables	PC
heap	stack
open files	state
child processes	...
accounting information	
signals	
...	

For example...

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

Why threads?

- multithreaded applications can run faster on computers with multiple CPUs and/or cores
- multiple threads can parallelize access to hardware, eg. 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 downloading large file in one tab, while playing movie in another
- multithreaded design can sometimes be simpler than alternatives (non-blocking I/O, FSM)
- threads are "lighter weight" compared to processes
  - cheaper to create and destroy (10-100 times faster)
  - take up less memory
  - access to more efficient communication mechanisms via shared memory
  - context switch can be more efficient

Why not threads?

- if a thread crashes, the whole process could crash
- programming with threads is harder than with processes, we have to worry about things like: race conditions, deadlocks, livelocks, starvation, …

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
- ...

Thread example: web server

- Requests for pages come in and the requested page is sent back to the client.
- Tasks such as receiving requests from the network interface, fetching the requested page from the disk, or sending the page to the network interface, are all I/O bound.
- We need to serve as many requests per second as possible.

How can threads help?

- option 1 – not the best
  - one thread for receiving the requests
  - one thread for sending the pages
  - one thread for fetching the page from the disk
- option 2 – much better
  - create separate thread for each request… problems?

Common thread scenarios

- **pipeline**
  - a task is broken into a series of stages
  - each stage handled by a different thread
- **manager/worker** (aka master/slave)
  - one manager thread assigns work to worker threads
  - manager thread handles all I/O
  - worker threads can be static or dynamic
- **peer**
  - all threads work on the same or different tasks in parallel

Thread pool

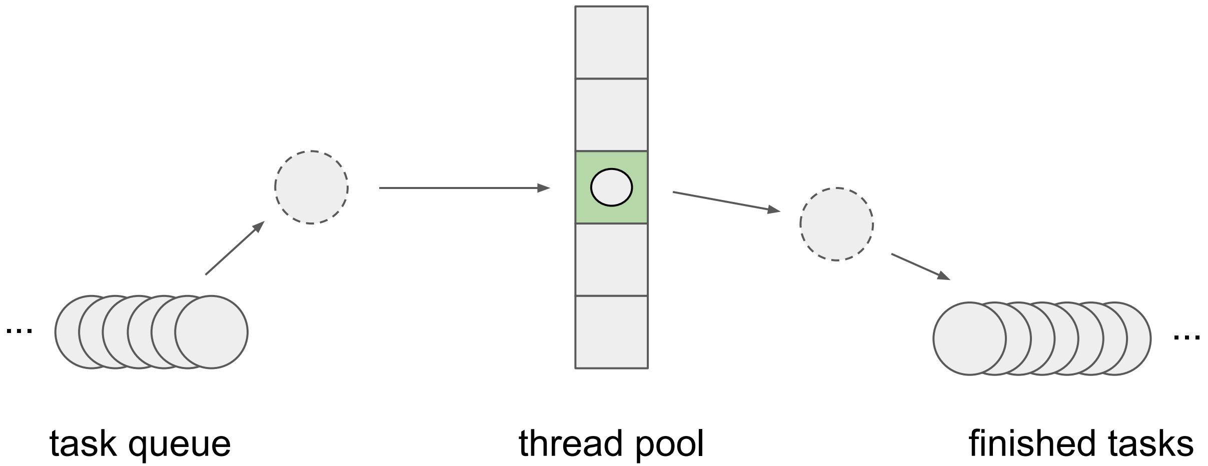
- recall the web-server example
  - when server receives a request, it creates a separate thread to handle the request
  - once request handled, thread is destroyed
- problems:
  - frequent thread creation and termination → performance problem
  - potentially large number of concurrent threads → resource problem
- solutions ?

Thread pool

- **thread pool** – a software design pattern
- program creates and maintains a pool of worker threads
- pool size can be tuned, eg. to the available computing resources
- when program needs a thread, it is takes one out of the pool
- when thread is done, program returns the thread back to the pool (**thread recycling**)
- benefits:
  - thread creation/destruction costs are reduced
  - number of possible concurrent threads is limited
- problems:
  - what if the program needs more threads than the size of the pool?
- solutions ?

Thread pool + task queue

- thread queues are often combined with a **task queue**
- instead of asking for a thread, a 'task' is inserted into a task queue
- available threads in the thread pool take tasks from the task queue, and finish them
- task queue can even be augmented to support multiple priorities…
  - but beware of possible dependencies between tasks



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
  - Win32
  - Java

POSIX threads (aka pthreads)

- to use POSIX threads
  - `#include <pthread.h>`
  - compile with `-lpthread`
- **`pthread_create`**`(thread, attr, start_routine, arg)`
  - starts a thread, similar to `fork()`
- **`pthread_exit`**`(status)`
  - terminates the current thread, similar to `exit()`, or you can return from `start_routine`
- **`pthread_join`**`(thread, * 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

Processes & global variables


```
#include <stdio.h> <stdlib.h> <unistd.h> <sys/wait.h>

int x;

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

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

```
Output:

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

Processes & global variables

```
#include <stdio.h> <stdlib.h> <unistd.h> <sys/wait.h>

int x;

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

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

Output:

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

https://repl.it/Lulm/1

Threads & global variables

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

int x;

void * do_something(void * ) {
  x = 11;
  pthread_exit(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:

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

Compile with:

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

Threads & global variables

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

int x;

void * do_something(void * ) {
  x = 11;
  pthread_exit(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:

```
$ ./a.out
x = 11
```

https://repl.it/LuoF/0

```c
#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];
  long status, i;
  for (i = 0; i < NUMBER_OF_THREADS; i++) {
    printf("creating thread %ld\n", (long int) i);
    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);
  exit(0);
}
```

Example with multiple threads

Compile with:

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

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

Homework

- 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 odd numbers 1, 3, 5, …
    - stores result in one global variable
  - thread 2:
    - calculates sum of even numbers 2, 4, 6 …
    - stores result in another global variable
- main thread
  - parses command line argument & starts 2 threads
  - waits for both threads to finish
  - sums the two global variables & prints out the result

Signal handling

- signal handling is more complicated with threads
  - which thread should handle the signal?

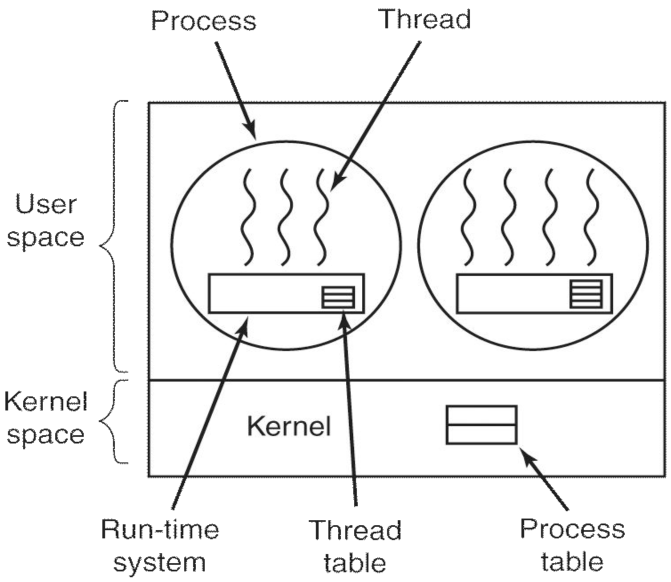
    i.e. in which thread's context should the signal handler be executed?
  - what about user-level threads?
- in POSIX systems, signal delivery depends on the type of the signal:
  - some signals are thread specific:
    - eg. `SIGSEGV` is delivered to the thread that caused the exception
    - `pthread_kill(thread_id, 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 implementations

- kernel-level threads
  - managed by the kernel/OS
  - most common
- user-level threads
  - entirely implemented in user space, usually as a library
  - kernel knows nothing about threads
  - not very common, used in some HPC environments for efficiency
- hybrids

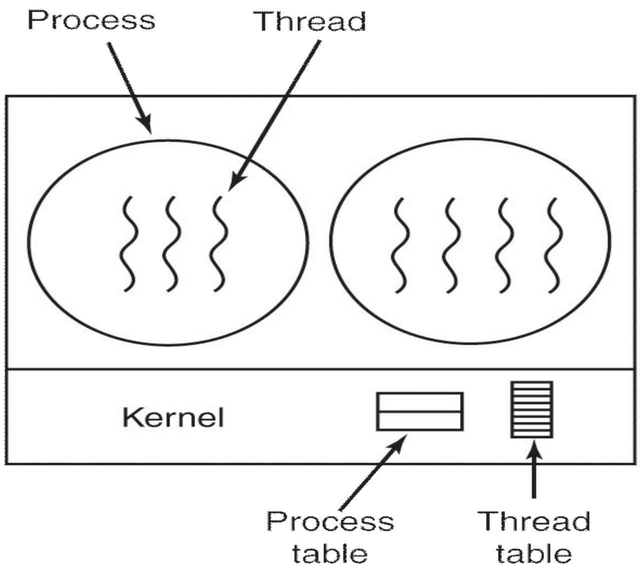
User-level threads

- 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

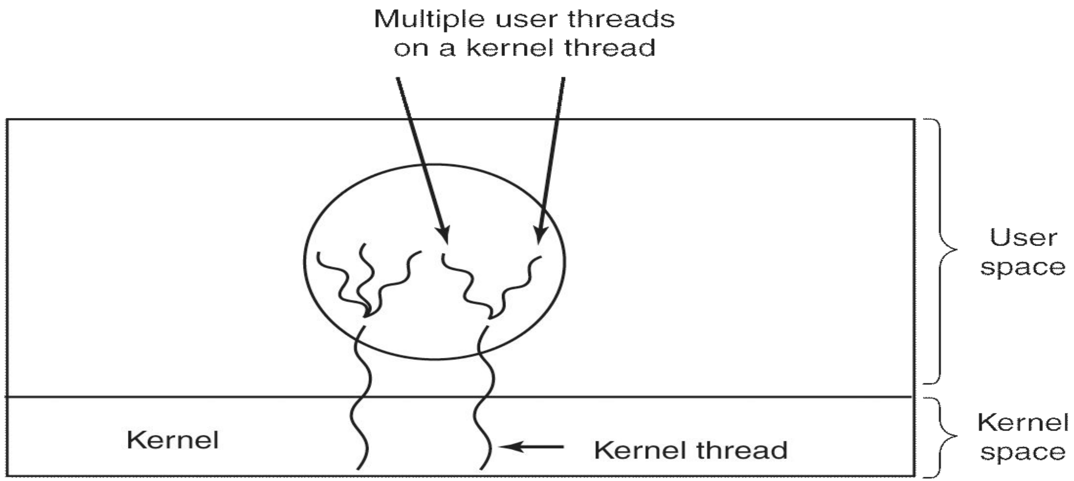


User-level vs kernel-level threads

	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

- 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

- 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

Summary

Reference: 2.1.2 - 2.1.5, 2.2.1-2.2.5 (Modern Operating Systems)

3.1 - 3.3, 4.1-4.3 (Operating System Concepts)

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.

Questions?