Introduction to threads
What are threads?

- Threads are also called **lightweight processes**

- Conventional processes are called **heavyweight processes**

- Some definitions:
  - Threads are “mini” processes that run within the memory space of one process.
  - Threads are independent, concurrent *instruction execution paths* within one process
Memory layout of a multi-threaded process

- Threads will dynamically use up stack and heap space. Every new allocation (of new data or new function call) will be attributed to the calling thread, so threads have their own heap and stack, but all are within the same address space!
(Heavyweight) Processes vs Threads

- **Processes**
  - Separate memory space
  - They are visible with OS utilities, e.g. `ps` and `top`.

- **Threads**
  - Common memory space
    - Communication of data among threads is more convenient
  - Faster to create (20×) and switch (6×) than processes
Benefits of multi-threading vs...

Single-threaded implementations
- Responsiveness
  - In single-threaded, if there is a blocking I/O, then all of the process is blocked
- Concurrency
- Other...?

Multi-process implementations
- Sharing name space of variables (ease of programming)
- Economy
  - Sharing same memory space (memory savings)
  - Don’t have to allocate space multiple times
- Scalability through economy
- Other...?
POSIX Threads (Pthreads)

- **pthread_create()**
  - Creates a thread

- **pthread_attr_init()**
  - Initializes the attributes (permissions, priority, inheritance of environment etc) of a thread to be created

- **pthread_exit()**
  - Terminates a thread (from within)

- **pthread_cancel()**
  - Terminates a thread (from outside)

- **pthread_join()**
  - Waits for a thread to exit/return
POSIX threads programming example

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

void *thread_main(void *);

int main()
{
    pthread_t tid;
    pthread_attr_t attribs;
    pthread_attr_init(&attribs);
    pthread_create(&tid, &attribs, thread_main, NULL);
    return 0;
}

void *thread_main(void * arg)
{
    printf("Hello, threaded world!");
    pthread_exit(0);
}
```
Multi-threading Issues
User threads vs kernel threads

- **User threads**
  - Threads that run at **user space**
  - They are the visible threads to a programmer
  - To exploit parallelism/concurrency by users

- **Kernel threads**
  - Threads spawned by the kernel that run in **kernel space**
  - To exploit parallelism/concurrency at kernel level
  - Not visible to user processes
Many-to-one (M:1)

- All threads of a process are mapped to a single thread in the kernel
- If one thread blocks, all "tethered" threads block
- Implementations:
  - GNU threads
  - Green threads in Solaris
One-to-one (1:1)

- For each thread in user space, there is a corresponding thread in kernel space
- Implementations
  - Linux
  - Windows
Many-to-many (M:M)

- M user threads map to “N” kernel threads.
- Allows to limit the kernel threads
- How is the user-to-kernel thread mapping done?
  - Static mapping?
    - At its creation, a thread is statically assigned to a kernel thread
  - Dynamic mapping?
    - On each context switch, a thread may be assigned to a different thread
- Implementations
  - Solaris versions < 9
Thread safety (thread-safe functions)

- Thread-safe definition
  - A function that may be safely invoked concurrently by multiple threads. Each function defined in the System Interfaces volume of POSIX.1-2008 is thread-safe unless explicitly stated otherwise. Examples are any “pure” function, a function which holds a mutex locked while it is accessing static storage, or objects shared among threads. [POSIX Std 1003.1-2008]

- How to ensure thread safety of “non-pure” functions
  - Use mutual exclusion mechanisms
  - Properly divide work to achieve separation of data
  - other...?
fork() and exec() behavior

- What happens when fork() is called?
  - Are the threads mirrored? Are the re-initialized?

Solutions

- Ad-hoc

- What happens when exec() is called?
  - What if other threads are “in the middle of doing something” (which is more than often the case)?
Thread cancellation

- Cancellation
  - Terminating a thread from outside the thread
  - Analogous to “process killing”
- Asynchronous Cancellation
  - Thread is immediately canceled irrespective of its state
- Deferred Cancellation
  - Thread is canceled only at points in the execution path that are safe to perform the cancellation
    - Cancellation points
Handling process signals

• Issue
  • What thread is accounted/charged for the signal handler when a signal is being delivered to a process?

• Options
  • Deliver the signal to “the obviously related” thread
  • Deliver the signal to every thread or many threads
  • Deliver the signal to a specific thread that is assigned to receive signals