Processes vs. Threads

A problem with processes
- It is expensive to create a new process:
  - construct new data structure in the kernel
  - copy memory
- It is somewhat expensive to switch between processes (context switch):
  - cache must be flushed
  - registers reloaded
  - kernel scheduler and dispatcher is always involved

Threads
- Multiple “threads” of execution within a single address space.
- Also known as “lightweight processes”
- Thread creation is 10-100 times faster than processes.

Threads
- Threads in the same process share:
  - process instructions
  - most data
  - open files (descriptors)
  - signals and signal handlers
- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
Thread models

- Threads in a single kernel process:
  - usual pthreads implementation
  - thread scheduling is done in an application library
  - if a thread blocks the whole process blocks

- One thread per kernel thread
  - kernel is now involved in scheduling (a little more expensive to create and switch between threads)
  - easier to take advantage of SMPs

Creating a thread

```c
int pthread_create(pthread_t *tid,  
  pthread_attr_t *attr,  
  void *(*func)(void*), void *arg);
```

- `tid` uniquely identifies a thread within a process and is returned by the function
- `attr` sets attributes such as priority, initial stack size
  - can be specified as NULL to get default
- `func` - the function to call to start the thread
- `arg` is the argument to `func`
- returns 0 is successful, a positive error code if node does not set `errno` but returns compatible error codes
- can use `strerror()` to print error messages

Creating threads

- There is no notion of parent and child, only thread pools.

Waiting for a thread to terminate

```c
int pthread_join(pthread_t tid, void **status)
```

- `tid` - the tid of the thread to wait for
- cannot wait for any thread (as in `wait()`)
- `status`, if not NULL returns the void * returned by the thread when it terminates.
- a thread can terminate by
  - `returning from func`
  - the main() function exiting
  - `pthread_exit()`

Other pthread functions

```c
void pthread_exit(void *status)
```

- a second way to exit, returns status explicitly
- status must not point to an object local to the thread, as these disappear when the thread terminates.

```c
void pthread_detach(pthread_tic);
```

- threads are either joinable or detachable
- if a thread is detached its termination cannot be tracked with `pthread_join()` - it becomes a daemon thread
- `pthread_t pthread_self(void)`
- returns the thread ID of the thread which called it
- you will often see `pthread_detach(pthread_self())`

Passing Arguments to Threads

```c
pthread_t thread_ID;
int fd, result;
result = pthread_create(&thread_ID,  
  (pthread_attr_t *)NULL,  
  myThreadFcn, (void *)&fd);
if(result != 0)  
  printf("Error: %s\n", strerror(result));
```

- We can pass any variable (including a structure or array) to our thread function. It assumes the thread function knows what type it is. The above example is bad if the main thread alters `fd` later.
Standard Technique

- Use `malloc()` to create memory for the argument to pass
- `init` var's value
- pass pointer to new memory via `pthread_create()`
- thread function releases memory when done.

typedef struct myArg {
    int fd;
    char name[25];
} MyArg;

int result; pthread_t thread_ID;
MyArg *p = (MyArg *)malloc(sizeof(MyArg));
strcpy(p->name, "CSC209", 7);
result = pthread_create(&thread_ID, NULL, myThreadFcn,
(void *)p);

void *myThreadFcn(void *p){
    MyArg *theArg = (MyArg *) p;
    write(theArg->fd, theArg->name, strlen(theArg->name));
    close(theArg->fd); free(theArg);
    return NULL;
}