Tasks and Multithreading

Introduction

Tasks are a great tool to do multiple things at once, but they can be difficult to use properly. Each task has a *priority* and a *stack size*. The higher the priority, the more crucial the task is considered, and more CPU time will be awarded to the task. Tasks of higher priority will *always* run in preference to lower priority tasks, unless the higher priority task is using `delay()` or some other waiting action.

Tasks are creating using `taskCreate()`, which invokes a user function in the new task:

```
void myFirstTask(void * parameter) {
  while(true) {
    printf("Hello from another task!\n");
    delay(500);
  }
}

void initialize() { // or some other function
  TaskHandle firstTaskHandle = taskCreate(myFirstTask, TASK_DEFAULT_STACK_SIZE, NULL, TASK_PRIORITY_
  delay(20000); // or doing anything else
  taskDelete(firstTaskHandle); // stop running the task
}
```

Periodic functions

If you want a function to run every *n* milliseconds, you can use `taskRunLoop`. The example above can be implemented slightly differently and receive the same effect:

```
void myFirstRunLoop() {
  printf("Hello from another task!\n");
}

void initialize() {
  TaskHandle secondTaskHandle = taskRunLoop(myFirstRunLoop, 500);
  delay(20000); // or doing anything else
```

```
  taskDelete(secondTaskHandle);
}
```

Best Task Practices

- Limit the number of tasks; five medium sized tasks will run quicker than ten tiny ones. There is also a limit of twelve tasks running at once.
- Tasks are usually not automatically stopped if the robot is disabled, unlike `operatorControl` and `autonomous`. Tasks running while disabled cannot use the VEX Joystick or VEX Motors. If the task should stop when the robot is disabled, use the `isOnline` and/or the `isAutonomous` function to control the program accordingly. If `taskRunLoop` is used, the task will automatically be cancelled if the robot is disabled or switched between driver and autonomous.
- Most tasks should have a priority from `TASK_PRIORITY_LOWEST + 1` to `TASK_PRIORITY_HIGHEST - 1`. Tasks of the lowest or highest priority may cause unexpected behavior.
- Tasks should wait when there is no work to do so other tasks can run. If a task, especially a high-priority one, does not occasionally use delay or similar, other tasks may not run or run very slowly. Since hardware such as the VEX LCD and Integrated Motor Encoders is also updated by tasks, a run away task can cause unexpected behavior.
- The Cortex Microcontroller is very fast. It is unlikely that any repetitive task will gain from running continuously. Even in the fastest scenarios, `delay(2)` will probably make no noticable difference, but will prevent future task starvation issues.
- Tasks using motors or joysticks should use `delay(20)` as the motors and joysticks are only updated every 20 ms.

Synchronization

One problem which one often runs into when dealing with tasks is the problem of synchronization. If two tasks try to read the same sensor or control the same motor at the same time, unexpected behavior may occur since two tasks are trying to read/write to the same piece of data.

Tasks can be designed to never conflict over motors or sensors: (division of responsibility)

```
void Task1(void * ignore) {
  // update motors 2 and 4
}

void Task2(void * ignore) {
```

```
  // update motors 5 and 6
}
```

Sometimes this is impossible: suppose you wanted to write a PID controller on its own task and you wanted to change the target of the controller. PROS features two types of synchronization structures, *mutexes* and *semaphores* that can be used to coordinate tasks. Mutexes stand for mutual exclusion; only one task can hold a mutex at any given time. Other tasks must wait for the first task to finish (and release the mutex) before they may continue.

```
Mutex mutex = mutexCreate();

// Acquire the mutex; other tasks using this command will wait until the mutex is released
// timeout can specify the maximum time to wait, or MAX_DELAY to wait forever
// If the timeout expires, "false" will be rturned, otherwise "true"
mutexTake(mutex, timeout);
// do some work
// Release the mutex for other tasks
mutexGive(mutex);
```

Semaphores are like signals - one task can take a semaphore to wait for a coordination signal from another task which gives the semaphore. Multiple tasks may wait for a semaphore; if this is the case, the highest priority task will continue per signal given.

```
// Create a semaphore
Semaphore semaphore = semaphoreCreate();

// Waits for the semaphore to be signalled
// timeout can specify the maximum time to wait, or MAX_DELAY to wait forever
// If the timeout expires, "false" will be rturned, otherwise "true"
semaphoreTake(semaphore);
// do something
// Signal the semaphore
semaphoreGive(semaphore);
```