# CS 241: Intro to Threads

Today, we are going to be discussing threads and their usefulness along with their applications.

### Terminal Trick of the Day!

Tab complete: If you're typing the name of a command or file in your shell, press tab to automatically complete as much of its name as possible. If multiple files share a prefix, press tab twice to show a list of matches. For example, if you wanted to find your Extreme Edge Cases MP in your SVN, and had typed `cd ~/cs241/e`, pressing tab would complete it to `cd ~/cs241/extreme_edge_cases/`, and pressing tab again would show files in that directory.

### Fixing Non-Reentrant Code

Change this code so that it is thread-safe.

```
char* perror_r(char *what) {
    static char buffer[4096], errname[100];
    int written = snprintf(buffer, 4096, "%s:%s", what,
        errname);
    write(2, buffer, written);
}
```

In multithreaded code, there is a strong notion of ownership when it comes to memory and memory errors. Many of the functions and data structures you will be writing will return a pointer and remove the data from the data structure entirely. If we didn't have this sense of "a thread owns this piece of memory", what would happen? (Hint: think of the example that we just did.)

### Splitting Work Up

If you look at today's lab, it involves an *highly parallel problem*. What does that mean?

Let's say that we have an array that we want to split up between threads. Each thread will compute this loop.

```
for (int j = left_boundary; j < right_boundary; ++j) {
    do_someting_with_element(A[j]);
}
```

Complete this function that will divide the work up so that all of the array elements are split as equally as possible and so that **none of the elements overlap and all elements get covered**. (Hint: why may one be different?)

```
worklist_t *split_work(int thread_num, int num_threads,
    int array_len) {
    worklist_t work = malloc(sizeof(worklist_t))
    if (thread_num ==  ) {

        work->left_boundary =

        work->right_boundary =

    }
    else {

        work->left_boundary =

        work->right_boundary =

    }
    return work;
}
```

## Trace the Threads

Draw out the thread-join diagram for the following code (just like the process diagram from a couple weeks ago).

```c
void* work(void *data){
    sleep(1);
    if(data)
        pthread_join((pthread_t)data, NULL);
    return NULL;
}
int main(){
    pthread_t thread = (pthread_t)NULL;
    for(int i = 0; i < 5; ++i){
        pthread_create(&thread, NULL, work,
            (void*)thread);
    }
    pthread_join(thread, NULL);
}
```