Peterson Report

This is a very simple implementation of Peterson's algorithm using C and pthreads.

Initialization

The main program spawns two threads, one which will be printing 'a', the other 'b':

```
pthread_create(&threads[ids], NULL, t_func, (void *)ids);
```

Each thread will run `void *t_func(void *tid)` concurrently.

After the threads are finished executing, they are joined using:

```
pthread_join(threads[ids], &status);
```

Thread Function

```
void *t_func(void *tid)
```

Each thread executes this function, which prints a character depending on the calling thread. The thread's id is passed in as an argument.

Peterson's algorithm is used to synchronize access to a character count variable so that 30 characters are printed before a new line is printed. Currently, a global total of 1200 characters will be printed (40 lines).

Critical Section

Each thread prints a single character and updates a count of the number of characters printed thus far. Since this count is shared, updating it must be atomic. In addition, printing a character must also be a part of this atomic update, to avoid situations where one thread prints the 30th character in a line, and before the count is updated, another thread prints the 31st character.

```
/* start critical section */

if (count % 30 == 0) {

fprintf(stdout, "%c", "\n");

} else {

fprintf(stdout, "%c", printchar[id]);

fflush(stdout);

}
```

```
count++;

/* end critical section */
```

For my code, I use count % 30 instead of resetting count to zero so that a maximum number of characters are printed.

Analysis of Peterson's Algorithm for Mutual Exclusion

Peterson's algorithm uses two global variables to synchronize two threads:

```
bool flag[2] = {false, false};

int turn;
```

An outline of the algorithm, along with actual code:

(each algorithm executes the following behavior) 1. indicate that the current thread wishes to enter the critical section 2. give priority to the other thread

```
flag[id] = true;

turn = other_id;
```

3. if the other thread is in its critical section, wait until it exits

```
while(flag[other_id] && turn == other_id) { ...wait... };
```

4. execute critical section
5. indicate that the current thread has left the critical section

```
flag[id] = false;
```

Mutual Exclusion Enforcement

Peterson's algorithm enforces mutual exclusion at step 3. In particular, if one thread wishes to enter the critical section, it will only do so if either the other thread does not want to enter its own critical section, or if it is the current thread's turn.

Process Halts in Noncritical Section

This is no problem for Peterson's algorithm, as one thread will only wait for another thread if said thread has indicated that it wishes to enter its critical section, i.e. when it sets `flag[id] = true`.

Indefinite Delay

Assuming no thread blocks inside the critical section (see Critical Section Finite), then a waiting thread will enter the critical section as soon as the other leaves. Even in the case where a thread leaves the critical section and then immediately requests access to it again, the algorithm assures that it will grant priority to any other threads that requested access before it did.

Immediate Critical Section Access

If there is only one thread that wishes to enter the critical section, then only its `flag[id]` will be true; all others will be false. Therefore, its wait condition will evaluate to false, and it will enter the critical section immediately.

No CPU Assumptions

The algorithm takes into account the number of concurrent threads that may wish to access a shared resource, however it does not rely on any knowledge about CPU speed or number of processors.

Critical Section Finite

In our implementation, the critical section only performs a finite number of tasks and none of those tasks rely on nondeterministic external processes. It would certainly be possible to add a call in the critical section that blocks indefinitely, which is something that must be avoided.