High Performance Computing: Sheet 4

Nils Döring Michael Mardaus Julian F. Rost
Sebastian Müller

4. Dezember 2013

Question 1

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define WORKLOAD 128

int done=0;
int current_job=0;
void* (*jobs[WORKLOAD])( int input );

pthread_mutex_t mutex;
pthread_cond_t awaken;

// input will be the job's id
void* job( int input )
{
        printf( "%10s(): starting number %d\n", __FUNCTION__, input );
        // SUBSTITUTE YOUR JOB BELOW HERE
        sleep(1);// (simulates "work" being done)
        // SUBSTITUTE YOUR JOB ABOVE HERE
        printf( "%10s(): completed number %d\n", __FUNCTION__, input );
}

void generate( int position );
void* loop( void* id );

int main( int argc, char** argv )
{
        // basic initialization of variables
        int i;
        int t=atoi( argv[1] );
        if( t<1 )
        {
                exit(EXIT_FAILURE);
        } else if( t > WORKLOAD )
        {
                t=WORKLOAD;
        }
```

```
        pthread_t threads[t];

        // additional initialization
        pthread_mutex_init( &mutex, NULL );
        pthread_cond_init( &awaken, NULL );

        // create t threads and make them wait for some work
        for( i=0; i<t; ++i )
        {
                pthread_create( &threads[i], NULL, loop, (void*)i );
                printf( "%10s(): created thread %d/%d\n", __FUNCTION__, i, t );
        }

        // generate WORKLOAD many jobs
        for( i=0; i<WORKLOAD; ++i )
        {
                generate( i );
                printf( "%10s(): generated job number %d\n", __FUNCTION__, i );
        }

        // tell all threads to awaken until all jobs have been completed
        while( current_job<WORKLOAD-1 )
        {
                pthread_cond_broadcast( &awaken );
        }
        done=1;

        // join all threads after their work is done
        for( i=0; i<t; ++i )
        {
                pthread_join( threads[i], NULL );
                printf( "%10s(): joined thread %d\n", __FUNCTION__, i );
        }
        printf( "%10s(): waited on %d threads -- done\n", __FUNCTION__, t );

        // cleanup duty -- somebody's gotta do it
        pthread_mutex_destroy( &mutex );
        pthread_cond_destroy( &awaken );
        pthread_exit( NULL );
}

void generate( int position )
{
        jobs[position]=&job;
        pthread_mutex_lock( &mutex );
        printf( "%10s(): signalling job %d is ready\n", __FUNCTION__, position );
        pthread_cond_signal( &awaken );
        pthread_mutex_unlock( &mutex );
}

void* loop( void* id )
{
        int my_id=(int)id;
```

```
        printf( "%10s(): starting in thread %d\n", __FUNCTION__, my_id );
        pthread_mutex_lock( &mutex );
        while( !done )
        {
                printf( "%10s(): thread %d waiting for work\n", __FUNCTION__, my_id );
                pthread_cond_wait(&awaken, &mutex);
                printf( "%10s(): thread %d received \"awaken\" signal\n", __FUNCTION__, my_id );
                pthread_mutex_unlock( &mutex ); // get read to do the job (in parallel)
                jobs[current_job++]( current_job ); // do the actual job
                pthread_mutex_lock( &mutex ); // wait for another job (serially)
                printf( "%10s(): thread %d going back to sleep\n", __FUNCTION__, my_id );
        }
        pthread_mutex_unlock( &mutex );
        pthread_exit( NULL );
}
```