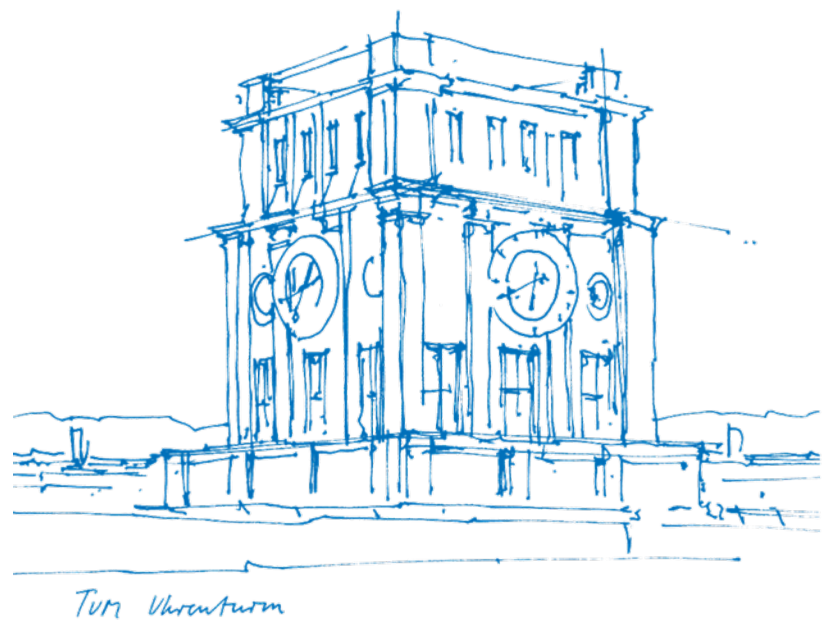
Parallel Programming Tutorial - Pthread 3

M.Sc. Andreas Wilhelm

Technichal University Munich

May 02, 2016



Organization

- There is no tutorial next week and the week after (May, 09 and 16)
- Deadline for Assignment 2 is on May 23, 3:30 pm
- The assignment will be online on Wednesday, May 04
- The slides will be updated then

Assignment 1: Possible Solution (Speedup 2,38) (1/3)

1. Parallelization of `playEM()`
   - `playEM()` creates `numThreads` threads
   - The calls of `playGroup()` are evenly distributed throughout the threads
   - The arrays `teams`, `successors`, and `bestThirds` are given as input to each thread
   - There is no race condition since each thread is operating on disjunct data
   - `bestThirds` are sorted after the threads have been joined

```c
struct pthread_args
{
  int groupMin;  // start index of the group number
  int groupMax;  // end index of the group number
  int numSuccessors;
  int teamsPerGroup;
  team_t* teams;
  team_t** successors;
  team_t** bestThirds;
};
```

Assignment 1: Possible Solution (Speedup 2,38) (2/3)

```c
// compute number of groups for each thread (plus a rest)
int g = 0
int groupsPerThread = NUMGROUPS / numThreads;
int groupsRest = NUMGROUPS % numThreads;

// assign the groups to numThreads threads
for (t = 0; t < numThreads; ++t) {
  thread_arg[t].groupMin = g;
  g += groupsPerThread;
  if (groupsRest > 0) { g++; groupsRest--; }
  thread_arg[t].groupMax = g;
  thread_arg[t].teams = teams;
  thread_arg[t].numSuccessors = numSuccessors;
  thread_arg[t].teamsPerGroup = teamsPerGroup;
  thread_arg[t].successors = successors;
  thread_arg[t].bestThirds = bestThirds;
  pthread_create(threads+t, NULL, &parallel_calls_to_playGroup, thread_arg+t);
}
```

```c
void* parallel_calls_to_playGroup(void * ptr) {
  struct pthread_args *arg = ptr;
  int g;

  for (g = arg->groupMin; g < arg->groupMax; ++g) {
    playGroup(g, arg->teams + (g * arg->teamsPerGroup),
                 arg->teamsPerGroup,
                 arg->successors + g * 2,
                 arg->successors + (arg->numSuccessors - (g * 2) - 1),
                 arg->bestThirds + g);
  }
  return NULL;
}
```

Assignment 1: Possible Solution (Speedup 2,38) (3/3)

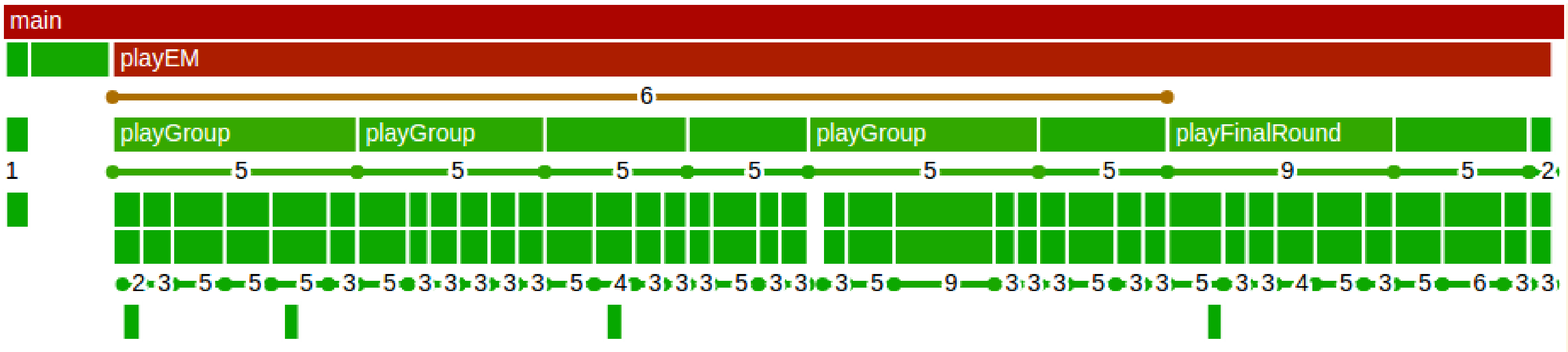
2. Parallelization of `playFinalRound()`
   - Each call of `playFinalRound()` creates `numSuccessors` threads
   - Each thread calls `playFinalMatch()` for a single match
   - Input is the number of games, the current game number, and the team information
   - The resulting number of goals is written to the pointer argument by each thread

```c
struct pthread_args_match {
  int numGames;
  int gameNo;
  team_t* team1;
  team_t* team2;
  int goals1;
  int goals2;
};
```

```c
for (i = 0; i < numGames; ++i) {
  args[i].numGames = numGames;
  args[i].gameNo = i;
  args[i].team1 = teams[i*2];
  args[i].team2 = teams[i*2+1];
  pthread_create(threads + i, NULL, &playMatchInPar, &args[i]);
}
```

```c
void* playMatchInPar(void* ptr) {
  struct pthread_args_match *args = (ptr);
  playFinalMatch(args->numGames, args->gameNo,
                 args->team1, args->team2, &args->goals1, &args->goals2);
  return NULL;
}
```

Assignment 1: Fast solution (Speedup 3,41) (1/2)



1. Parallelization of `playEM()`
   - Idea: avoid load imbalance by a clever distribution of the `playGroup()` calls
   - Thread 1: `playGroup(0)`
   - Thread 2: `playGroup(1, 3)`
   - Thread 3: `playGroup(2, 5)`
   - Thread 4: `playGroup(4)`

Assignment 1: Fast solution (Speedup 3,41) (2/2)

2. Parallelization of `playFinalMatch()` calls
   - The results for all games (group and final games) are deterministic
   - Idea: play the final matches in parallel to the group games by additional 4 threads
   - Postpone the join of all threads to the end of `playEM()`

```c
void* parallel_calls_to_playFinalMatch(void* arg){
  struct pthread_args_final* args = arg;
  int goals1, goals2, i;

  for (i = 0; i < args->threadGames; ++i) {
    playFinalMatch(args->numGames, args->gameNo,
                   args->team1, args->team2,
                   &goals1, &goals2);
    if (goals1 == goals2)
      playPenalty(args->team1, args->team2, &goals1, &goals2);
    args++;
  }
  return NULL;
}
```

```c
finalArgs[0].threadGames = 4;
finalArgs[0].numGames = 8;
finalArgs[0].gameNo = 0;
finalArgs[0].team1 = teams + 0;
finalArgs[0].team2 = teams + 1;
// ...
finalArgs[12].threadGames = 3;
finalArgs[12].numGames = 2;
finalArgs[12].gameNo = 0;
finalArgs[12].team1 = teams + 0;
finalArgs[12].team2 = teams + 8;
```

Condition Variables

Definition

A Condition Variable (CV) is the mechanism your program waits for a predicate to become true, and to communicate to others that it might be true

- Used for communicating information about shared state data
- Some threads may **wait** for some predicate to become true
- Some threads may **signal** a single waiting thread
- Some threads may **broadcast** all waiting threads
- Example: Signal that a queue is no longer empty
- Every condition variable is associated with:
  - a mutex (for synchronization)
  - a predicate (the shared state data)

Condition Variables: Initializing

```c
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init ( pthread_cond_t *cond,
                        pthread_condattr_t *condattr );
int pthread_cond_destroy( pthread_cond_t *cond );
```

- pthread_cond_t *cond
  - Pointer to a condition variable
- pthread_condattr_t *condattr
  - Optional pointer to a pthread_condattr_t struct to define behaviour, if NULL defaults are used
- Use static initialization for static CV's with default attributes (you do not have to destroy a static CV)
- Use dynamic initialization for malloc and non-standard attributes (destroying of the CV necessary)

Condition Variables: Waiting (1/2)

```c
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
int pthread_cond_timedwait( pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            struct timespec *expiration );
```

- pthread_mutex_t *mutex
  - Pointer to the mutex that is associated with cond
- struct timespec *expiration
  - The timespec structure that sets the max. expiration time
- Before waiting on a CV, the mutex must always be locked
- Waiting on a CV unlocks the mutex and waits on a signal/broadcast (atomically!)
- On signal/broadcast, the mutex will be locked again

Condition Variables: Waiting (2/2)

```c
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
int pthread_cond_timedwait( pthread_cond_t *cond,
                            pthread_mutex_t *mutex,
                            struct timespec *expiration );
```

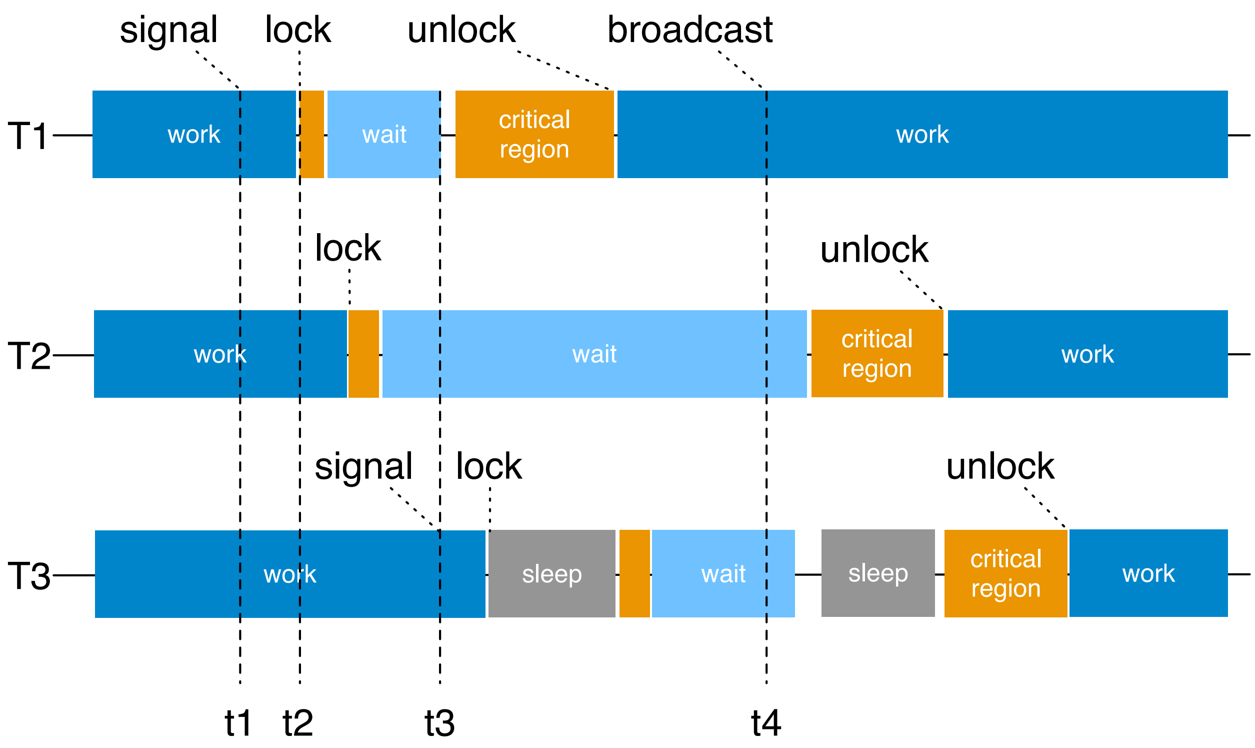
- Always check your predicate before waiting and then check it again (in a loop) to prevent:
  1. Intercepted Wakeups
     Other threads may wake up earlier and grab the work. Never assume that the last `pthread_cond_wait` gets the lock!
  2. Loose Predicates
     Signaling thread may use a weak predicate ("there may be work" instead of "there is work"). You have to check for strict predicate again!
  3. Spurious Wakeups
     Wait may wakeup without signal/broadcast!

Condition Variables: Waking up Waiters (1/2)

```c
int pthread_cond_signal( pthread_cond_t *cond );
int pthread_cond_broadcast( pthread_cond_t *cond );
```

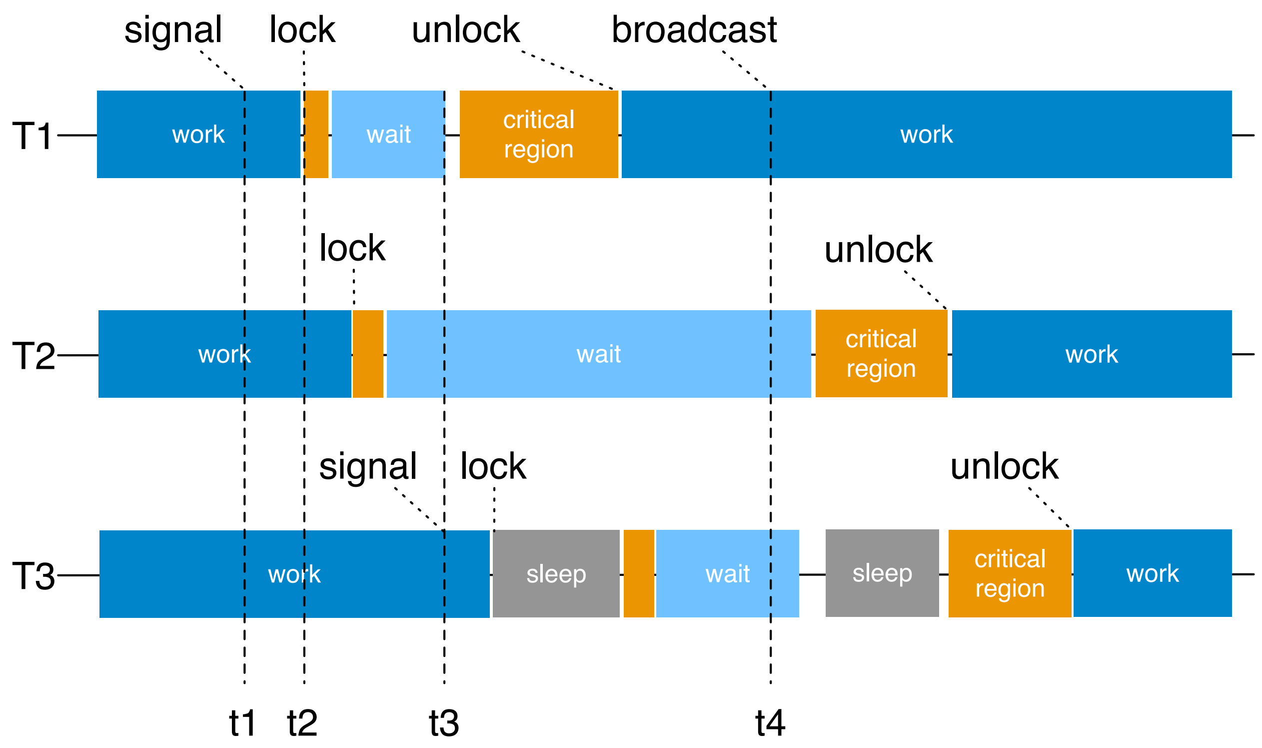
- signal wakes up a single thread waiting on cond
- broadcast wakes up all threads waiting on cond
- signal/broadcast without any waiters has no effect
- signal is more efficient (less context switches)
- signal/broadcast while holding the mutex may infer additional context switches, but may prevent intercepted wakeups

Condition Variables: Illustration



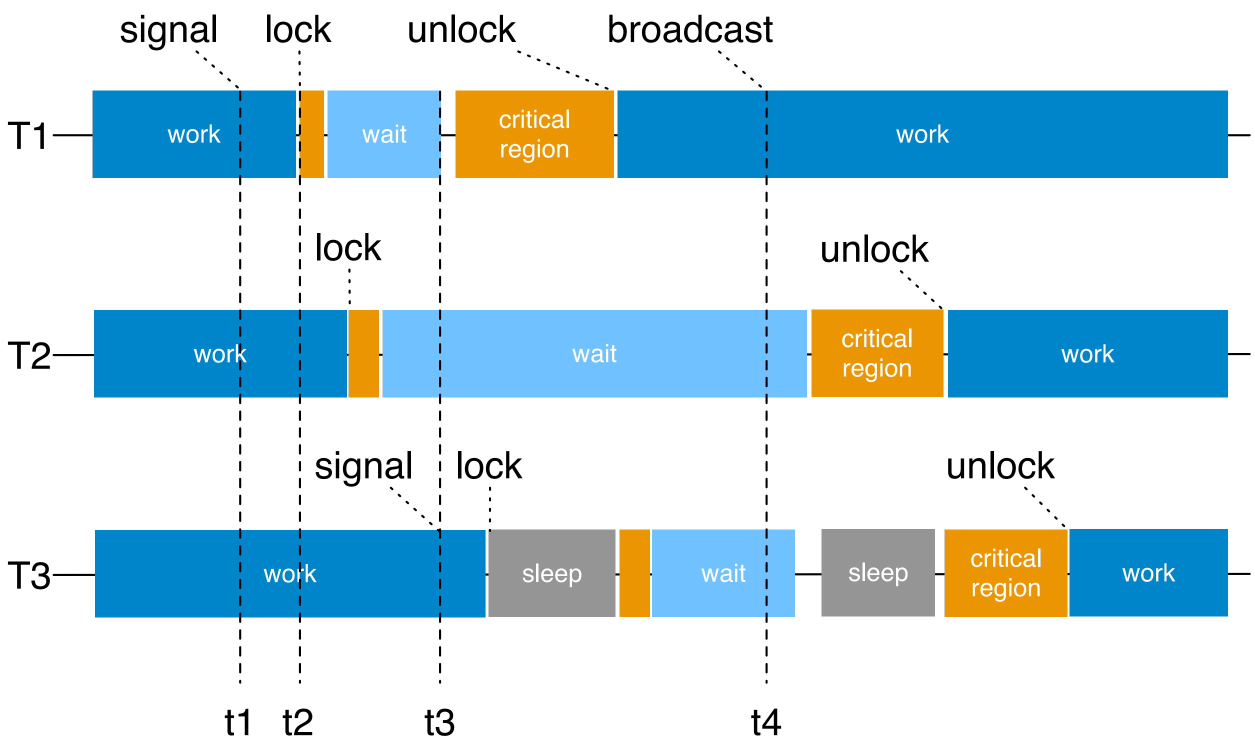
- **t1:** T1 signals with no waiters

Condition Variables: Illustration



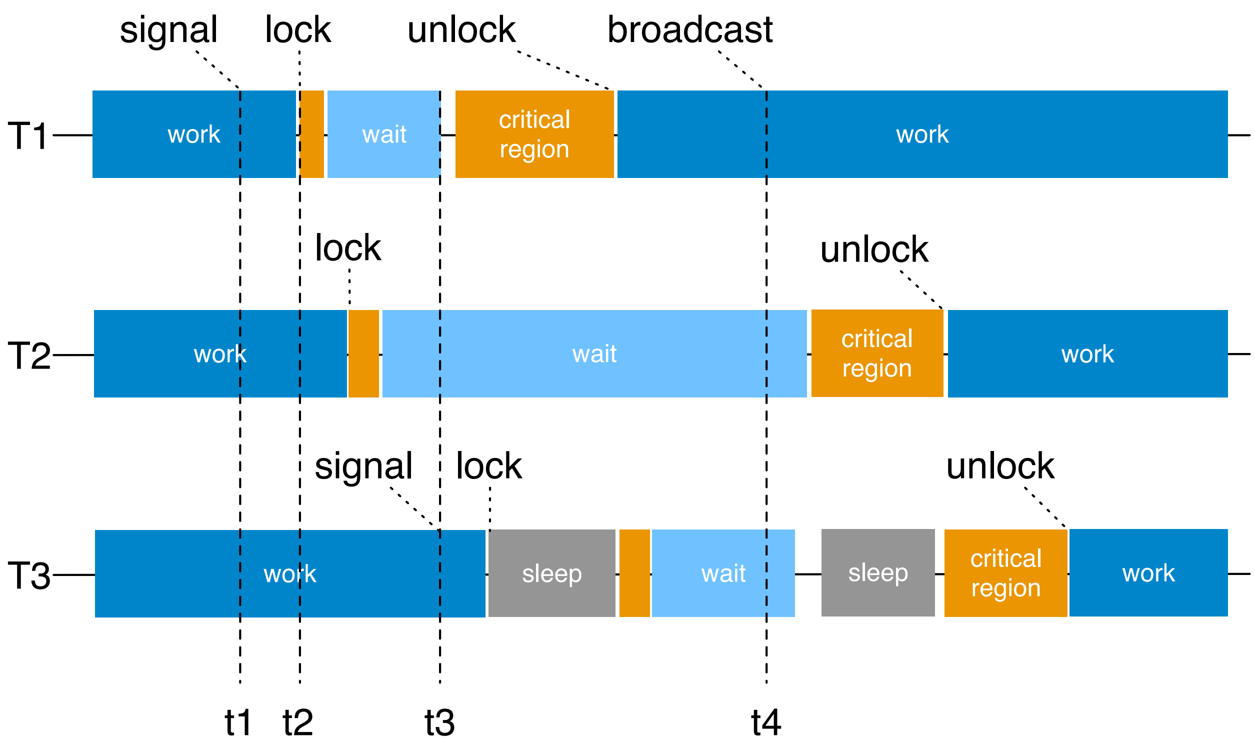
- **t2:** T1 waits for a predicate to hold. Therefore, it first locks a mutex, then checks the predicate (false), and then calls pthread_cond_wait (which unlocks the mutex and waits)

Condition Variables: Illustration



- **t3:** T3 signals and wakes up T1, which locks the mutex and works on the critical region.

Condition Variables: Illustration



- **t4:** T1 broadcasts and wakes up T2 and T3.

Condition Variables: Example (1/2)

```c
#include <stdio.h>
#include <pthread.h>
#include <time.h>
#include <errno.h>

typedef struct {
  pthread_mutex_t   mutex;
  pthread_cond_t    cond;
  int               value;
} my_struct_t;

my_struct_t data = { PTHREAD_MUTEX_INITIALIZER,
                     PTHREAD_COND_INITIALIZER,
                     0 };

int hibernation = 1; // Default to 1 second

void *signal_thread( void *arg ) {

  sleep( hibernation );
  pthread_mutex_lock( &data.mutex );
  data.value = 1; // set predicate
  pthread_cond_signal( &data.cond );
  pthread_mutex_unlock( &data.mutex );

  return NULL;
}
```

Condition Variables: Example (2/2)

```c
int main( int argc, char**argv ) {

  pthread_t thread;
  struct timespec timeout;

  pthread_create( &thread, NULL, &signal_thread, NULL);
  clock_gettime(CLOCK_REALTIME, &timeout);
  timeout.tv_sec += 2;   // wait for predicate for 2 seconds

  pthread_mutex_lock( &data.mutex );
  while( data.value == 0) { // important!
    if (pthread_cond_timedwait( &data.cond, &data.mutex, &timeout ) == ETIMEDOUT) {
      printf( "Condition wait timed out.\n");
      break;
    }
  }
  if (data.value != 0) // consider the timeout
    printf( "Condition was signaled.\n");
  pthread_mutex_unlock( &data.mutex );

  pthread_join( thread, NULL );
  return 0;
}
```

Pthreads: What is missing?

- Special attributes for threads, mutexes, etc.
- Thread cancelation
- Thread-specific data
- Scheduling
- Priority-aware mutexes
- Barriers, Semaphores, Read-Write-Locks, etc.

Assignment 2: Dynamic EMSim (1/2)

Task

- Use POSIX threads to parallelize `playGroups()` and `playFinalRound()` of `emsim_seq.c`
- The program should follow the producer/consumer pattern:
  - There is one producer thread (the main thread) that produces game information and writes it to a buffer (for group games and final games).
  - There are `numWorker` worker threads that use the game information as input and play the matches asynchronously.
  - As soon as match information is available on the buffer, a free worker grabs it and begins to play.
  - The results are used by the producer thread to progress.
- Consider:
  - This time, `playGroups()` will be called once for all 36 matches of the group phase.
  - The number of created worker threads N is checked (`numWorker` <= N <= 2 x `numWorker`).
  - You may have to use global storage, think about synchronization.
  - You can solve the assignment with or without condition variables.
  - The speedup with 4 cores must be at least 3.
  - There is a new executable (`checks`) that you can use for valgrind/ltrace checks (use debug mode to get meaningful messages).

Assignment 2: Dynamic EMSim (2/2)

Hints for using condition variables

- Use two condition variables:
  - One CV for the producer (consumer ready)
  - One CV for the consumer (work available)
- Use a single mutex for both condition variables.
- The producer needs to wait for the consumers to finish their work at the end.
- Remember: If there is a signal and no one is waiting -> nothing happens.
- Never make a copy of a CV, only references allowed.

Assignment 2: emsim_seq.c (1/2)

```c
// play all (36) group games by utilizing numWorker worker threads
void playGroups(team_t* teams, int numWorker) {
  static const int cNumTeamsPerGroup = NUMTEAMS / NUMGROUPS;
  int g, i, j, goalsI, goalsJ;

  for (g = 0; g < NUMGROUPS; ++g) {
    for (i =  g * cNumTeamsPerGroup; i < (g+1) * cNumTeamsPerGroup; ++i) {
      for (j = (g+1) * cNumTeamsPerGroup - 1; j > i; --j) {

        // team i plays against team j in group g
        playGroupMatch(g, teams + i, teams + j, &goalsI, &goalsJ);
        teams[i].goals += goalsI - goalsJ;
        teams[j].goals += goalsJ - goalsI;
        if (goalsI > goalsJ)
          teams[i].points += 3;
        else if (goalsI < goalsJ)
          teams[j].points += 3;
        else {
          teams[i].points += 1;
          teams[j].points += 1;
        }
      }
    }
  }
}
```

Assignment 2: `emsim_seq.c` (2/2)

```c
// play a specific final round by utilizing numWorker worker threads
void playFinalRound(int numGames, team_t** teams, team_t** successors, int numWorker){
  team_t* team1;
  team_t* team2;
  int i, goals1 = 0, goals2 = 0;

  for (i = 0; i < numGames; ++i) {
    team1 = teams[i*2];
    team2 = teams[i*2+1];
    playFinalMatch(numGames, i, team1, team2, &goals1, &goals2);

    if (goals1 > goals2)
      successors[i] = team1;
    else if (goals1 < goals2)
      successors[i] = team2;
    else {
      playPenalty(team1, team2, &goals1, &goals2);
      if (goals1 > goals2)
        successors[i] = team1;
      else
        successors[i] = team2;
    }
  }
}
```

Assignment: Dynamic EMSim - Provided Files

- Makefile
  - contains rules to build executables
  - available targets: parallel, sequential, unit_test, checks, all (default), clean
  - 'mode=debug make [target]' to build debug version, use 'make clean' before
- main.c
  - main function - argument handling + build teams + call playEM
- emsim.h
  - Header file for emsim.c and emsim_*.c
- emsim.c
  - Defines the simulator logic
- db.h / db.c
  - Header and definition for the database accesses
- emsim_seq.c
  - Sequential version of `playGroups()` and `playFinalRound()`.
- student/emsim_par.c
  - Implement the parallel version in this file

Assignment: Dynamic EMSim - Provided Files

- em.db
  - Input data: The database containing all em results.
- libsqlite3.a
  - the slite3 library to read the database
  - there is also a libsqlite3_32.a (in case you have a 32bit system) -> in that case, you have to modify the Makefile
- vis.h / vis.c
  - The visualization component
- unit_test.c
  - The unit tests that execute both the serial and parallel version to compare results.