MULTITHREADING* PART 2

Ana Lucia VARBANESCU
a.l.varbanescu@uva.nl
SNE, IvI, UvA

Some images and text courtesy of LLNL, USA: https://computing.llnl.gov/tutorials/pthreads/#Abstract

2 lectures on parallel processing

- Last Tuesday …
  - Basics and terminology
  - Multi-threading
  - High-level multithreading: OpenMP

- Today …
  - OpenMP *briefly* revisited
  - Low-level threading libraries: POSIX Threads
  - Models of parallel computation (if time permits)

Quiz results

- Parallelize RGB to gray-scale
  - Explain how you parallelize, what does each thread do, and what is the expected performance …
- In general
  - Everybody split the image across threads
    - Different ways: squares, rows, …
  - Some chose threads = cores, other selected a fixed number
  - Very few discussed performance
  - Advice:
    - Design rarely has a single solution – make sure you explicitly state your assumptions and you understand why you make certain decisions.

Gathering points …

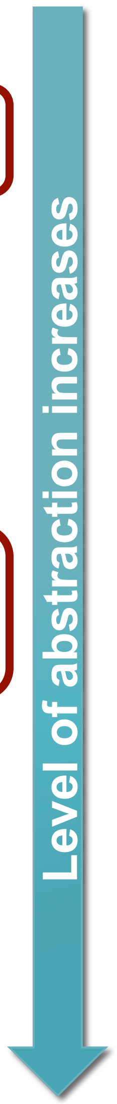
- Quizzes = in-class activity
  - 5-10p each, depending on difficulty
- Optional assignments = at home, with given deadline
  - ~10p each

- Total ~ 80p

- >50p => 1 full point added to your exam grade
- >20p & <80p => scaling

Programming models

- Pthreads
- TBB – Thread building blocks
  - Threading library
- OpenCL
  - To be discussed …
- OpenMP
  - High-level, pragma-based
- … <many more>
- Cilk
  - Simple divide-and-conquer model
- … <many more>



OPENMP

Final remarks

OpenMP reminder

- Pragma-based model
  - Start with sequential code
  - Add pragma’s to indicate parallelism
- Advantages
  - High productivity
  - (deceivingly) Easy to use
- Disadvantages
  - Still lots of responsibility for the programmer, sometimes hidden
  - Can be difficult to debug
  - Can lack flexibility

OpenMP in a nutshell

- Parallelism :

  `#pragma omp parallel [clauses]`
- Work sharing

  `#pragma omp for [clauses]`
- Challenges
  - Variable scope
  - Reduction
  - Synchronization
  - Load balancing
  - …
- Tasks

Example: can we use OpenMP?

```
double x, y;
int i, j, max = 200;
int depth [M,N];
...
for (i =0; i<M; i++) {
  for (j =0; j<N; j++) {
        x=(double) i / (double) M;
        y=(double) j / (double) N;
        depth [i,j] = mandelval ( x, y, max );
} }
```

Properties to check:

- No data dependencies between loop iterations ?
- Trip-count known in advance ?
- Function mandelval without side-effects ?
- Only loop variable i needs to be private ?

Example: can we use OpenMP?

```
double x, y;
int i, j, max = 200;
int depth [M,N];
...
# pragma omp parallel for private (x, y, j)
for (i =0; i<M; i++) {
   for (j =0; j<N; j++) {
         x = ( double ) i / ( double ) M;
         y = ( double ) j / ( double ) N;
         depth [i,j] = mandelval ( x, y, max );
} }
```

- Added a clause => in this case, tag the specified variables as private.
- NOTE: Private variables are **not** initialised outside parallel section

*https://en.wikipedia.org/wiki/Mandelbrot_set

Shared and private variables …

- Shared variables:
  - One instance shared between sequential and parallel execution.
  - Value unaffected by transition.
- Private variables:
  - One instance during sequential execution.
  - One instance per worker thread during parallel execution.
  - No exchange of values.
- Firstprivate variables
  - Like private variables, but …
  - Worker thread instances initialised with master thread value.

How about data races?

```
int total = 0;
...
for (i =0; i<M; i++) {
  for (j =0; j<N; j++) {
      x = ( double ) i / ( double ) M;
      y = ( double ) j / ( double ) N;
      depth [i,j] = mandelval ( x, y, max );
      total = total + depth [i,j];
} }
```

Solution

```
int total = 0;
...
# pragma omp parallel for private (x, y, j)
for (i =0; i<M; i++) {
  for (j =0; j<N; j++) {
      x = ( double ) i / ( double ) M;
      y = ( double ) j / ( double ) N;
      depth [i,j] = mandelval ( x, y, max );
      # pragma omp critical
      {
            total = total + depth [i,j];
      }
}}
```

- The **critical** directive implements a critical region:
  - The directive must immediately precede new statement block.
  - Statement block is executed without interleaving.

Another solution: Reduction

Reduction clause

```
# pragma omp parallel for private ( x, y, i, j) \
                           reduction (+: total )
for (i=0; i<M; i++) {
  for (j=0; j<N; j++) {
     x = ( double ) i / ( double ) M;
     y = ( double ) j / ( double ) N;
     depth [i,j] = mandelval(x, y, max );
     total = total + depth [i,j];
 }
}
```

- Properties:
  - Reduction clause only supports built-in reduction operations: +, *, ^, &, |, &&, ||.
  - User-defined reductions only supported via critical regions.
  - Bit accuracy is not guaranteed.

Conditional parallelisation

```
if ( len < 1000)
  for (i =0; i< len ; i ++) {
    res [i] = a[i] * b[i];
  }
else
# pragma omp parallel for
  for (i =0; i< len ; i ++) {
    res [i] = a[i] * b[i];
  }
```

- **Or use the if clause**

```
# pragma omp parallel for if ( len >= 1000)
for (i =0; i< len ; i ++) {
res [i] = a[i] * b[i];
}
```

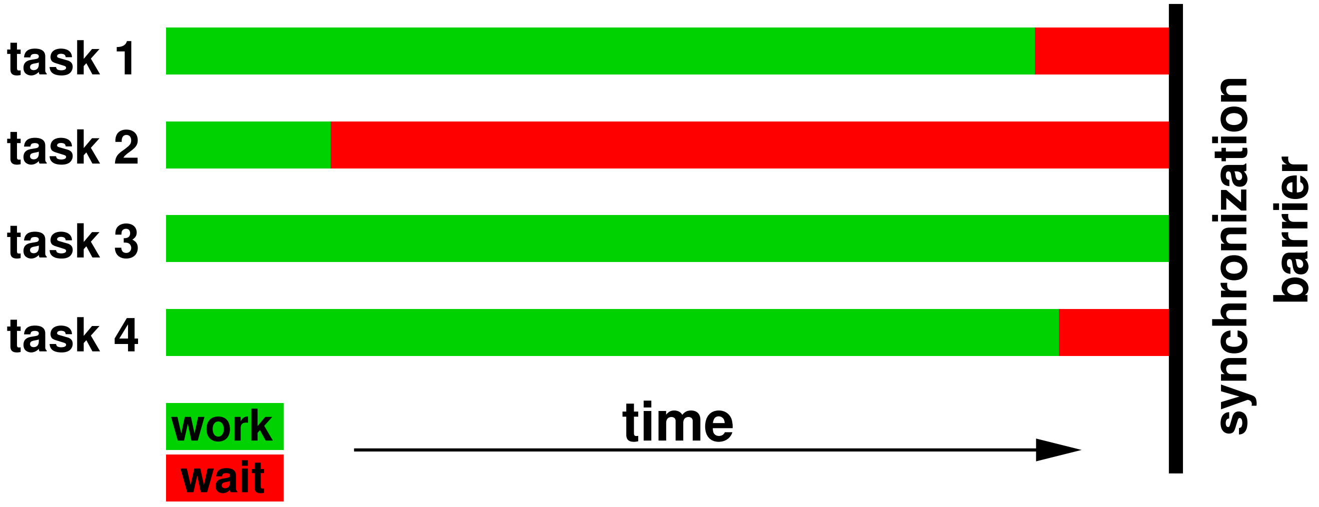
Loop scheduling

- **Definition:**

Loop scheduling determines which iterations are executed by which thread.

- **Aim:**

Equal workload distribution



Loop scheduling

Problem:

- Different situations require different techniques

The schedule clause:

```
# pragma omp parallel for schedule (<type> [,<chunk>])
for (...)
{ … }
```

- **Properties:**
  - Clause selects one out of a set of scheduling techniques.
  - Optionally, a chunk size can be specied.
  - Default chunk size depends on scheduling technique.

Loop scheduling

- Static: each threads gets a contiguous block
  - Aka, block scheduling
- Static with chunk size: chunks of P iterations until the full loop is finished.
  - Aka: block-cyclic scheduling
- Dynamic:
  - Loop is subdivided into chunks of P iterations (default P=1)
  - Chunks are dynamically assigned to threads on their demand.
  - Aka: self scheduling.
  - Problems:
    - More overhead
    - More synchronization

Loop scheduling

- Guided scheduling:

```
# pragma omp parallel for schedule ( guided , <n >)
```

- Chunks are dynamically assigned to threads on their demand.
  - Initial chunk size is implementation dependent.
  - Chunk size decreases exponentially with every assignment.
  - Aka: guided self scheduling.
  - Minimum chunk size: n (default: 1)

More on OpenMP: openmp.org

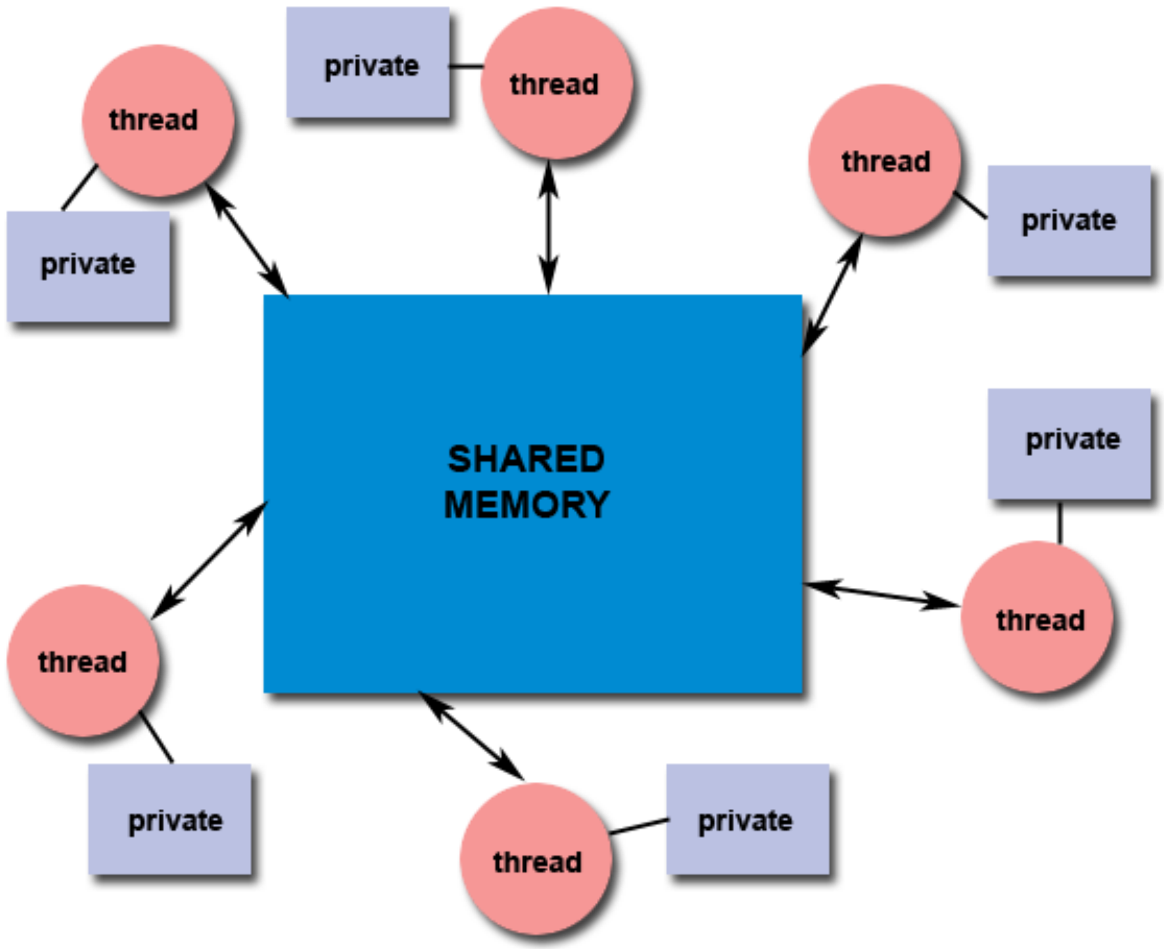
- More in OpenMP-2:
  - Decouple parallel regions from work sharing
  - Control synchronisation barriers
  - Task parallel sections
  - Low-level locks and condition variables
  - ...
- More in OpenMP-3:
  - Nested parallel regions
  - Spawning and synchronisation of tasks
  - ...
- More in OpenMP-4:
  - Support for accelerators
  - ...

pthreads (POSIX threads)

What are “Pthreads”

- Standardized programming interface for threads
  - UNIX systems : IEEE POSIX 1003.1c standard (1995)
- All implementation of these standards are known as POSIX threads or pthreads.
- Pthreads are defined as a set of C language programming types and procedure calls
  - pthread.h header/include file
  - a thread library
    - Or part of libc in some implementations.

Shared memory model



Types of subroutines

- Thread management
  - Includes creation and termination
- Mutexes (short for “mutual exclusion”)
  - Creating, destroying, locking, unlocking mutexes
- Conditional variables
  - Communications between threads that share a mutex.
  - Functions to set, query, and use conditional variables
- Synchronization
  - Manage read/write locks and barriers

Basic thread manipulation

- Thread creation

```
int pthread_create (
  pthread_t * thread_id,          // OUT: identifier
  pthread_attr_t * attributes, // IN: attributes
  void *(* start_routine )( void *), // IN: code to execute
  void *argument ) // IN: arguments for start_routine
```

- Waiting for a thread (join)

```
int pthread_join (
  pthread_t thread_id ,  // IN: thread to join
  void **return_value )  // OUT: return value of start
routine
```

Example: hello world [1]

```
int main () {
   pthread_t thread_ids[MAX];
   int i, *num;
   void *result;

for (i=0; i<MAX; i++) {
    num = (int*) malloc( sizeof( int));
    *num = i;
    pthread_create(&thread_ids[i], NULL,
         &HelloWorld, num);}
for (i=0; i<MAX; i++) {
    pthread_join(thread_ids[i], &result);
    free(result);}
return 0; }
```

Example: hello world [2]

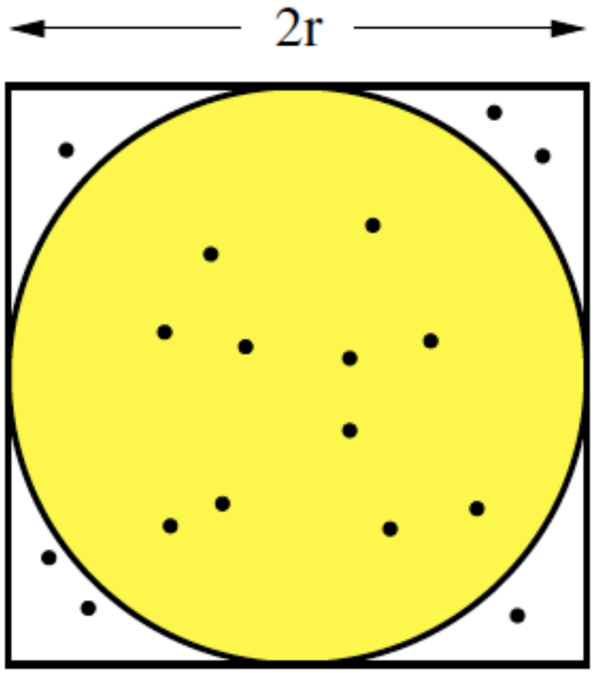
```
void *HelloWorld(void *a) {
   int* tid_p = (int*) a; // cast arg to int pointer
   int tid = *tid_p;      // dereference int pointer

   printf(”Hello World! I am thread %d !\n", tid);
   return a;
}
```

Example: Approximation of Pi

- 1. Inscribe circle of radius r in square of edge length 2r .
- 2. Randomly generate As points in the square.
- 3. Determine number of points Ac which are within circle.
- 4. Compute Pi:

  pi := 4 * (Ac/As)

- 5. More points, better approximation.
- 6. Known as Monte Carlo simulation.



Example: Approximation of Pi

```
double approx_pi ( int sample_points ) {
       int hits ; double pi;
       hits = simulate(sample_points);
       pi = 4.0 * ((double)hits) / ((double)sample_points);
       return pi;
}

int simulate ( int sample_points ) {
  int seed=1, hits=0, i;
  double rand_no_x, rand_no_y;
  for (i=0; i< sample_points; i++) {
   rand_no_x = (double)rand_r(&seed); //between 0 and 1
   rand_no_y = (double)rand_r(&seed); //between 0 and 1
   if (((rand_no_x-0.5)*(rand_no_x-0.5) +
        (rand_no_y-0.5)*(rand_no_y-0.5))< 0.25)
             hits ++;
   return hits ;
}
```

How do we do this with threads?

- Work per thread?
- Data distribution?
- Number of threads?

Parallel approximation of Pi [1]

```
double approx_pi (int num_threads, int sample_points ){
 int i, total_hits, hits[MAX_THREADS];
 pthread_t p_threads[MAX_THREADS];
 double pi;

 sample_points_per_thread = sample_points/num_threads ;
 for (i =0; i< num_threads ; i++) {
   hits[i] = i;
   pthread_create(&p_threads[i], NULL, simulate,
                        &hits[i]); }
all_hits = 0;
 for (i =0; i< num_threads ; i++) {
   pthread_join(p_threads[i], NULL );
   all_hits += hits[i]; }
pi = 4.0*((double)all_hits)/((double)sample_points );
return pi;}
```

Parallel approximation of Pi [2]

```
void* simulate(void *s) {
  int seed , i, *hit_pointer , local_hits ;
  double rand_no_x , rand_no_y ;
  hit_pointer = (int *)s;
  seed = *hit_pointer ;
  local_hits = 0;
  for (i=0; i<sample_points_per_thread ; i++) {
    rand_no_x=(double)rand_r(&seed)/(double)RAND_MAX;
    rand_no_y=(double)rand_r(&seed)/(double)RAND_MAX;
    if (((rand_no_x-0.5)*(rand_no_x-0.5)+
        (rand_no_y-0.5)*( rand_no_y-0.5))< 0.25) {
            local_hits ++; }}
  *hit_pointer = local_hits ;
  return NULL ;
}
```

Example: Approximation of Pi

```
double approx_pi ( int sample_points ) {
       int hits ; double pi;
       hits = simulate(sample_points);
       pi = 4.0 * ((double)hits) / ((double)sample_points);
       return pi;
}
int simulate ( int sample_points ) {
  int seed=1, hits=0, i;
  double rand_no_x, rand_no_y;
#pragma omp parallel for private(rand_no_x, rand_no_y)
                              reduction(+:hits)
  for (i=0; i< sample_points; i++) {
   rand_no_x = (double)rand_r(&seed); //between 0 and 1
   rand_no_y = (double)rand_r(&seed); //between 0 and 1
   if (((rand_no_x-0.5)*(rand_no_x-0.5) +
        (rand_no_y-0.5)*(rand_no_y-0.5))< 0.25)
             hits ++;
   return hits ;
}
```

Pthreads: Private vs. Shared Data

- Private Data:
  - Local variables of functions and blocks in general.
  - Exception: Their address is provided as thread argument.
  - Problem here: Stack frame of creator thread must survive created thread.
- Shared Data:
  - All global variables.
  - Any allocated heap structure provided its address is made accessible to multiple threads (e.g. via global variable)

Watch out for race conditions!

Critical region

Sequence of statements that MUST be executed without interleaving.

Thread 1:

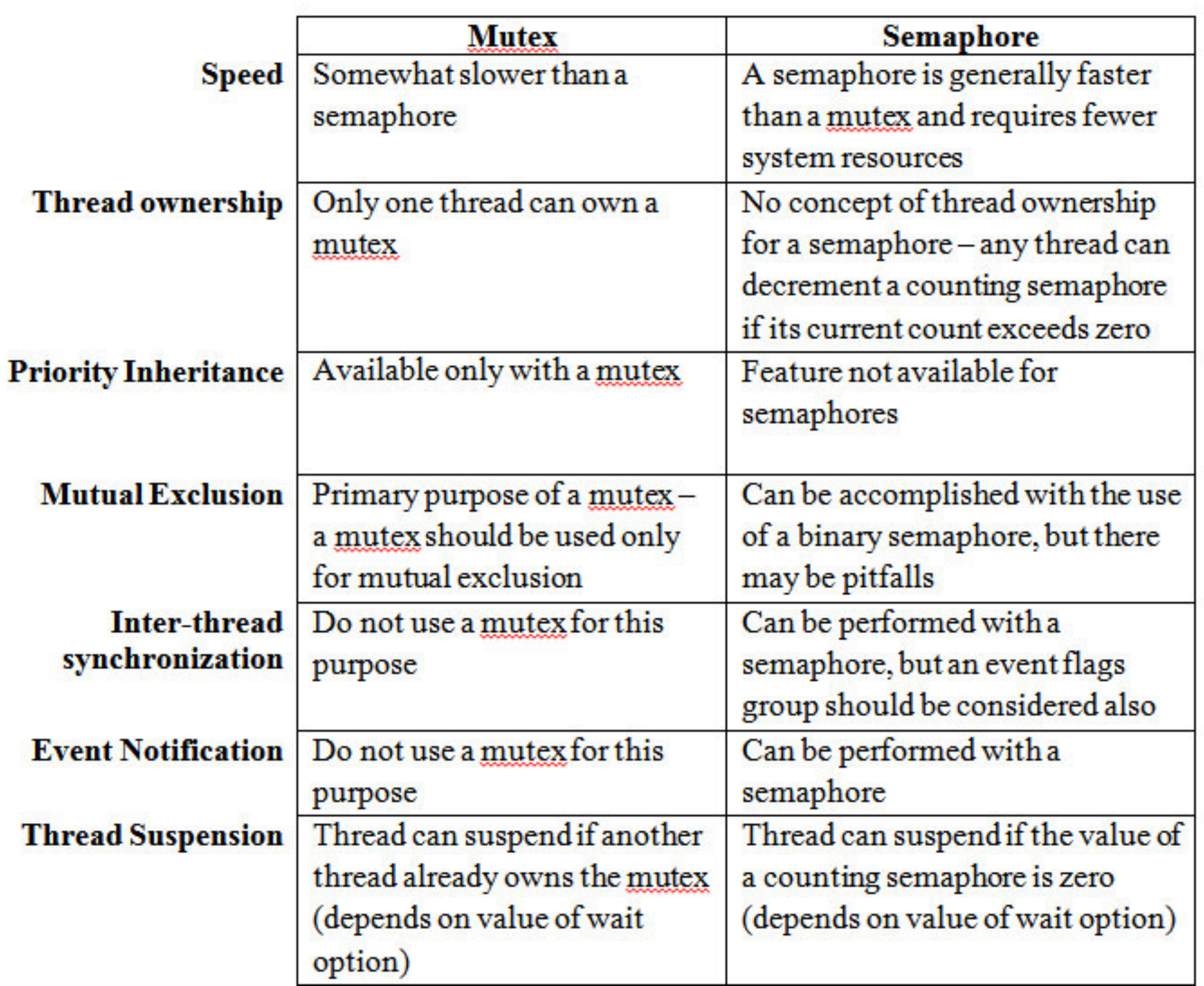
```
void isStudent() {
  if (person == student)
     critical {
        total++;
     }
}
```

Thread 2:

```
void isStudent() {
  if (person == student)
     critical {
        total++;
     }
}
```

Implementing critical regions

- Mutex locks
- Semaphores



	Mutex	Semaphore
Speed	Somewhat slower than a semaphore	A semaphore is generally faster than a mutex and requires fewer system resources
Thread ownership	Only one thread can own a mutex	No concept of thread ownership for a semaphore – any thread can decrement a counting semaphore if its current count exceeds zero
Priority Inheritance	Available only with a mutex	Feature not available for semaphores
Mutual Exclusion	Primary purpose of a mutex – a mutex should be used only for mutual exclusion	Can be accomplished with the use of a binary semaphore, but there may be pitfalls
Inter-thread synchronization	Do not use a mutex for this purpose	Can be performed with a semaphore, but an event flags group should be considered also
Event Notification	Do not use a mutex for this purpose	Can be performed with a semaphore
Thread Suspension	Thread can suspend if another thread already owns the mutex (depends on value of wait option)	Thread can suspend if the value of a counting semaphore is zero (depends on value of wait option)

Mutex locks: create & use

- Global:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- Per-thread:

```
pthread_mutex_lock ( & lock );
    …
    …
    …
 pthread_mutex_unlock ( & lock );
```

Critical region with mutex locks

```
//main program
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

//thread function
void *countStudent(…) {
  if (person == student) {
      pthread_mutex_lock(&lock);
          total++;
      pthread_mutex_unlock(&lock);
  }
}
```

Mutex locks

Properties:

- Mutex locks are abstract data objects.
- Only one thread at a time may hold a mutex lock.
- Threads block upon locking if lock is unavailable.
- Locking / unlocking are guaranteed to be atomic.
- No fairness on waiting threads upon unlocking.
- Only owner of lock can unlock.
- Re-locking by owner causes deadlock.

More mutex functions

- Dynamic initialisation with attributes:

```
int pthread_mutex_init ( pthread_mutex_t *lock ,
     pthread_mutexattr_t * attributes );
```

- Dynamic de-allocation:

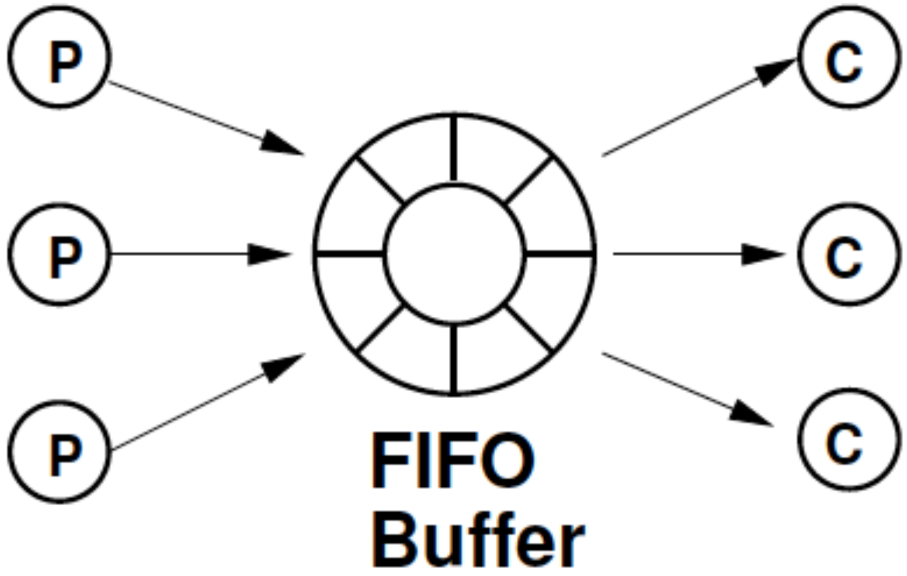
```
int pthread_mutex_destroy ( pthread_mutex_t * lock );
```

- Optimistic locking:

```
int pthread_mutex_trylock ( pthread_mutex_t * lock );
     // returns 0 upon success
     // returns EBUSY upon failure
```

One final example: producer-consumer

- Characteristics:
  - Multiple producer processes / threads.
  - Multiple consumer processes / threads.
  - Single shared FIFO buffer.
  - Producers write data to shared buffer.
  - Consumers read data from shared buffer.



Example

```
void Producer () {
data val;
bool done;
for (;;) {
  val = ... ;
  done = false;
  do{
pthread_mutex_lock(&lock);
  if (occupied < BUFSIZE) {
    buffer[in] = val;
    in = (in + 1) % BUFSIZE;
    occupied = occupied + 1;
    done = true; }
pthread_mutex_unlock(&lock);
} while (!done); }}
```

```
void Consumer () {
data val;
bool done;
for (;;) {
   done = false;
   do{
pthread_mutex_lock( &buflock)
  if (occupied > 0) {
    val = buffer[out];
    out = (out+1)%BUFSIZE;
    occupied = occupied - 1;
    done = true; }
pthread_mutex_unlock(&lock);
 } while (!done);
DoSomething( val); }
```

Naïve implementation?

What we have:

- Producers and consumers synchronize even if they access disjoint buffer areas.
- Producer threads repeatedly acquire buffer lock although buffer is still full.
- Consumer threads repeatedly acquire buffer lock although buffer is still empty.

What we need:

- Suspend producer threads on full buffer.
- Wake-up producer threads after consumer threads have removed data items.
- Suspend consumer threads on empty buffer.
- Wake-up consumer threads after producer threads have provided data items.

Condition variables

```
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
```

- Allow threads to self-suspend based on *data*

```
int pthread_cond_wait( pthread_cond_t *condvar,
                        pthread_mutex_t *lock)
```

- Allow threads to trigger other threads

```
int pthread_cond_signal(pthread_cond_t *condvar)
int pthread_cond_broadcast(pthread_cond_t *condvar)
```

- They implement *event-driven critical regions*

Assignment: benchmarking (10p)

- Benchmark* the two solutions for the producer-consumer problem (5p)
  - Naïve: using regular mutex only
  - Using condition variables

- Compare the observed performance and discuss the reasons for this (lack of) difference. (5p)

- *Benchmarking:
  - Formulate a hypothesis/goal
  - Design experiments for the goal
  - Describe the expected outcome
  - Comment on differences compared to the expected outcome

SUMMARY

OpenMP vs. Posix threads

- Different layers of abstraction:
  - pThreads
    - Close to machine
    - Difficult to write call
    - Everything is explicit
    - Increased control over behavior
  - OpenMP
    - Higher-level programming model
    - Very useful for loop parallelization and, in general, for data parallelism
    - More difficult for complicated control flow
    - Easy to write code
    - Many implicit assumptions