Synchronization (race conditions)

What is the output of the following program??

```c
dp = 0;
for (id = 0; id < 4; id++)
    create_thread (..., sum_computed_values, ...);

void sum_computed_values ( );
{  pdp = compute the value ( );
    dp += pdp;
}
```

- A critical section is a section of code that can be executed by one processor at a time (to guarantee mutual exclusion)
- locks can be used to enforce mutual exclusion

```
get the lock ;

dp += pdp ;
release the lock ;
```

Most parallel languages provide ways to declare and use locks or critical sections.

Multithreading: shared variables

- A variable declared within the thread function is private to that thread (cannot be accessed by other threads)
- A variable declared outside the thread function is shared and is accessible by all threads
- The race condition explained in the last slide results if two threads simultaneously update a shared variable.
- Note that, if threads execute on different CPUs, then a shared variable can be cached by multiple CPUs. This creates the cache coherence problems talked about in Section 5.

**Example:**

Initially pd = 0
Thread 1 executes pd=1
Thread 2 executes pd=0

The cache coherent problem occurs despite the fact that pd=1 and pd=0 are both atomic (non-divisible) operations.
Mutual Exclusion

• We need mutual exclusion in both parallel and serial programs (why?)

• Locks can be used to allow mutual exclusion, and hence provides a mechanism for exclusive access to shared data.

• Hardware support is needed to implement locks

• Locks can be implemented if a load-modify-store instruction, which executes atomically (indivisible), is provided.

• May use atomic swap operations (see next slide).

• In a bus-connected system, atomic operations on a variable \( x \) can be implemented by:
  1) enforcing cache coherence
  2) not releasing the bus before finishing the load-modify-store instructions.

• A processor that wants to access \( x \) will spin waiting for the bus release.

Mutual exclusion

• Basic Facility Required
  – Atomically retrieve and change a value (Read and Modify Instructions)
  – Atomic Reads (Load) and Atomic Writes (Stores) are not sufficient

• Example: Atomic Swap
  – Interchanges a value in a register for a value in memory
    • loads the value from a memory location into the register
    • stores the value in register into the memory location

• Atomic swap can be used to acquire a lock in memory (stored in location Lock)

```plaintext
// Assume (Lock = 0) means the lock is free
Lock:
   Put 1 in Register, R
   Atomic Swap (R, Lock)
   if (R = = 1) then try again
Unlock:
   Lock = 0
```
Barrier synchronization

- A barrier synchronization between N threads can be implemented using a shared variable initialized to N.
- When a processor reaches the barrier, it decrements the shared variable by 1 and waits (in a busy wait loop) until the value of the variable is equal to zero before it leaves the barrier.

- Need locks???

- What if there is no shared variables (distributed memory machines)?

- Can you synchronize using special hardware?

The Pthread API

(see https://computing.llnl.gov/tutorials/pthreads/)

- Pthreads has emerged as the standard threads API (Application Programming Interface), supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.
- Provides two basic functions for specifying concurrency:

```c
#include <pthread.h>

int pthread_create (pthread_t *thread_handle,
                    const pthread_attr_t *attribute,
                    void (*thread_function)(void *),
                    void *arg);

int pthread_join (pthread_t thread_handle,
                  void *ptr);
```
**Mutual Exclusion**

- Critical sections in Pthreads are implemented using mutex locks.
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- A thread entering a critical section first tries to get a lock. It goes ahead when the lock is granted.
- The API provides the following functions for handling mutex-locks:

  ```c
  int pthread_mutex_lock ( pthread_mutex_t *mutex_lock);
  int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
  int pthread_mutex_init ( pthread_mutex_t *mutex_lock,
                              const pthread_mutexattr_t *lock_attr);
  ```

  Can replace by NULL

---

**An Example (compute π)**

The value of PI can be calculated in a number of ways. Consider the following method of approximating PI:

- Inscribe a circle in a square
- Randomly generate points in the square
- Determine the number of points in the square that are also in the circle
- Let \( A_c/A_s \) be the number of points in the circle divided by the number of points in the square
- \( \pi \approx 4 \times \frac{A_c}{A_s} \)
- Note that the more points generated, the better the approximation
An Example (compute $\pi$)

```c
#include <sys/time.h>
#define MAX_THREADS 64
void *compute_pi ( void *);

int total_hits, sample_points, sample_points_per_thread, num_threads;

main ( ) {
...
}

void *compute_pi (void *s) {
...
}
```

struct arg_to_thread {int t_seed ; int hits ;}

main ( int argc, char argv[] )    {
    sample_points = atoi(argv[1]) ; /* first argument is the number of points */
    num_threads = atoi(argv[2]) ;   /* second argument is the number of threads*/

    int i ;
    pthread_t p_threads[MAX_THREADS];
    pthread_attr_t attr;
    double computed_pi;
    struct arg_to_thread my_arg[MAX_THREADS] ;

    pthread_attr_init (&attr);
```
An Example (compute $\pi$)

```
total_hits = 0;
sample_points_per_thread = sample_points / num_threads;

for (i=0; i< num_threads; i++) {
    my_arg[i].t_seed = i;  /* can chose any seed – here i is chosen*/
    pthread_create (&p_threads[i], &attr, compute_pi, &my_arg[i]);
}

for (i=0; i< num_threads; i++) {
    pthread_join (p_threads[i], NULL);
    total_hits += my_arg[i].hits;
}

computed_pi = 4.0*(double) total_hits / ((double) (sample_points));
```

An Example (compute $\pi$)

```
void *compute_pi (void *s) {
    struct arg_to_thread *local_arg;
    int seed, i, local_hits;
    double rand_no_x, rand_no_y;

    local_arg = s;
    seed = (*local_arg).t_seed;
    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 ++;  /* the generated sample is inside the circle*/
        seed *= i;
    }

    (*local_arg).hits = local_hits;
    pthread_exit (0);
}
```

Re-entrent function to generate a random number between 0 and RAND_MAX
Need to compile with "gcc -D_REENTRANT –lpthread"
An Example (compute $\pi$)

```c
void *compute_pi (void *s) {
    struct arg_to_thread *local_arg;
    int seed, i, local_hits;
    double rand_no_x, rand_no_y;
    local_arg = s;
    seed = (*local_arg).t_seed;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double) rand_r(&seed) / RAND_MAX;
        rand_no_y = (double) rand_r(&seed) / 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 ++; /* the generated sample is inside the circle*/
        seed *= i;
    }
    int pthread_mutex_lock (pthread_mutex_t *m_lock);
    (*local_arg).hits = local_hits; total_hits += local_hits;
    pthread_exit (0);
    int pthread_mutex_unlock (pthread_mutex_t *m_lock);
}
```

Re-entrent function to generate a random number between 0 and RAND_MAX
Need to compile with “gcc -D_REENTRANT -lpthread”

Multiprocessors connected by networks (Section 6.7)

- Each processor has private physical address space
- Hardware sends/receives messages between processors
Loosely Coupled Clusters

- Network of independent computers
  - Each has private memory and OS
  - Connected using I/O system (e.g., Ethernet or a switch)
- Suitable for applications with independent tasks
  - Web servers, databases, simulations, ...
- High availability, scalable, affordable
- Problem: Low interconnect bandwidth (compared to SMP)

- Grid Computing
  - Computers interconnected by long-haul networks (e.g., Internet)
  - Work units farmed out, results sent back
  - Can make use of idle time on PCs (e.g., PITTGRID)

Programming a distributed address space machine

- Assume that 10000 values are stored in the local memories of 16 processors such that 625 values are stored in A[0] … A[624] in the local memory of each processor.
- All variables are local variables (each processor has its own copy) – no shared variables.
- The function “send(x,p)” sends a message containing the value of x to processor p.
- The function “receive(x)” receives a message and puts the received value in x.

```c
Sum = 0;
for (i=0 ; i < 625 ; i++)
    Sum = Sum + A[i];
Half = 8;
for (i=0 ; i < 4 ; i++)
    { if (2*Half > Pn >= Half ) send (Sum, Pn – Half );
      if (Pn < Half ) { receive (remote_sum);
        Sum += remote_sum ; }
    Half = Half / 2; }
```

- Where is the global sum?
- The distribution of the initial data to the local memories is done either by the programmer or by the compiler.