CS528
Multi-threading and OpenMP

A Sahu

Dept of CSE, IIT Guwahati

Outline

- Thread Memory Model
- Thread Safety : 4 Classic cases
- Thread Synchronization: Example
- Thread Programming Model
  - Boss/Worker, Peers, Pipeline
- Thread Pooling: Example : Manual
- **Implicit/Auto Thread Pooling: OpenMP/Cilk**

Thread
Memory Model & Safety

Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared variables?
  - Answer not as simple as "global variables are shared" and "stack variables are private"
- Requires answers to the following questions:
  - What is the memory model for threads?
  - How are variables mapped to memory instances?

Threads Memory Model

- Conceptual model:
  - Each thread runs in the context of a process
  - Each thread has its own separate thread context
    - Thread ID, stack, stack pointer, program counter, condition codes, and general purpose registers
  - All threads share remaining process context
    - Code, data, heap, and shared library segments of process virtual address space
    - Open files and installed handlers

Threads Memory Model

- Operationally, this model is not strictly enforced:
  - Register values are truly separate and protected
  - But any thread can read and write the stack of any other thread
- *Mismatch between conceptual and operational model is a source of confusion and errors*

Threads Accessing other Thrd's Stack

```
char **ptr;  /* global */
int main(){
  int i; pthread_t tid;
  char *msgs[N] = { "Hello1", "Hello2" };
  ptr = msgs;
  for (i = 0; i < 2; i++)
    pthread_create(&tid,NULL,thread,(void )i);
    pthread_exit(NULL);
}
```

```
void *thread(void *vargp) {
    int myid = (int)vargp;
    printf("[%d]: %s \n", myid, ptr[myid]);
}
```

Peer threads access main thread's stack indirectly through global ptr variable

Shared Variable Analysis

Variableich instance	Ref by main thread?	Ref by peer thread 0?	Ref by peer thread 1?
`ptr`	`yes`	`yes`	`yes`
`i.m`	`yes`	`no`	`no`
`msgs.m`	`yes`	`yes`	`yes`
`myid.p0`	`no`	`yes`	`no`
`myid.p1`	`no`	`no`	`yes`

Answer: A variable x is shared iff multiple threads reference at least one instance of x. Thus:

- `ptr` and `msgs` are shared.
- `i` and `myid` are NOT shared.

Parallel Counter: without Lock

```
#define NITERS 100
int cnt = 0; /* shared */
int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, count, NULL);
  pthread_create(&tid2, NULL, count, NULL);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  if(cnt!=NITERS*2) printf("BOOM! cnt=%d",cnt);
    else printf("OK cnt=%d\n", cnt);
}
```

```
void *count(void *arg) {
for(int i=0;i<NITERS;i++) cnt++;
}
```

```
$./badcnt
BOOM! cnt=196
$./badcnt
BOOM! cnt=184
```

cnt should be 200

What went wrong?!

Thread Safety

- Functions called from a thread must be *thread-safe*
- There are four (non-disjoint) classes of thread-unsafe functions:
  - **Class 1: Failing to protect shared variables : L/UL**
  - Class 2: Relying on persistent state across invocations
  - Class 3: Returning pointer to static variable
  - Class 4: Calling thread-unsafe functions

Class 1: Failing to protect shared variables

- Fix: Use Lock and unlock semaphore operations
- Issue: Synchronization operations will slow down code
- Example: `goodcnt.c`

```
void *count(void *arg) {
for(int i=0;i<NITERS;i++)
      pthread_mutex_lock(&LV);
      cnt++;
      pthread_mutex_unlock(&LV);
} // LV is lock variable
```

Class 2: Relying on persistent state across multiple function invocations

- Random number generator relies on static state
- Fix: Rewrite function so that caller passes in all necessary state, ➔ Maintain Thread Specific State

```
int rand() {
    static uint next = 1;
    next = next*1103515245 + 12345;
    return (uint)(next/65536)% 32768;
}
void srand(uint seed) {
    next = seed;
}
```

Class 3: Returning pointer to `static` variable

- Fixes: 1. Rewrite code so caller passes pointer to `struct`, Issue: Requires changes in caller and callee
- *Lock-and-copy :* Issue: Requires only simple changes in caller (and none in callee), However, caller must free memory

```
struct hostent  *gethostbyname(
                          char* name){
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

Class 3: Returning pointer to `static` variable

```
struct hostent *gethostbyname_ts(char *p) {
  struct hostent *q = Malloc(...);
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;    /* copy */
  V(&mutex);
  return q;
}
```

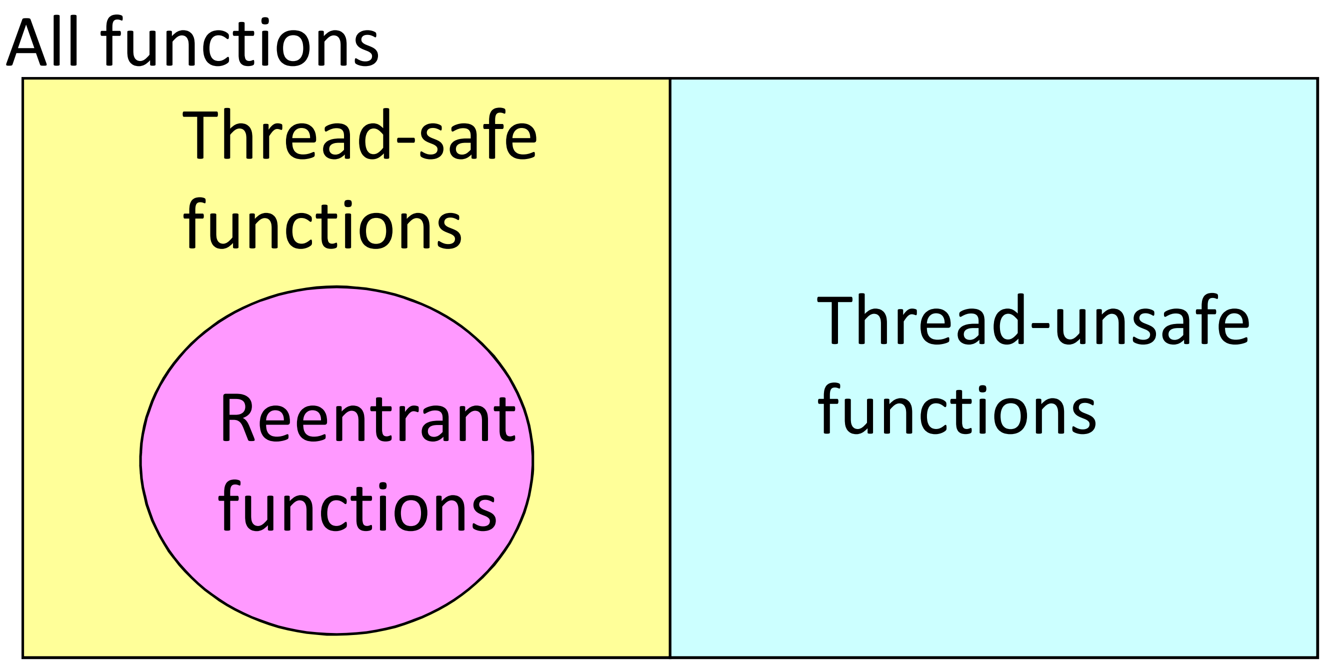
```
hostp = malloc(...));
gethostbyname_r(name, hostp);
```

Class 4: Calling thread-unsafe functions

- Calling one thread-unsafe function makes an entire function thread-unsafe
- Fix: Modify the function so it calls only thread-safe functions

Reentrant Functions

- A function is *reentrant* iff it accesses NO shared variables when called from multiple threads
  - **Reentrant functions are a proper subset of the set of thread-safe functions**
  - NOTE: The fixes to Class 2 and 3 thread-unsafe functions require modifying the function to make it reentrant (only first fix for Class 3 is reentrant)



Thread-Safe Library Functions

- Most functions in the Standard C Library (at the back of your K&R text) are thread-safe
  - Examples: **`malloc`**, **`free`**, **`printf`**, **`scanf`**
- All Unix system calls are thread-safe
- Library calls that aren't thread-safe:

Thread-unsafe function	Class	Reentrant version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	`(none)`
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

Synchronization : Lock/Unlock

Synchronization Hardware

- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words
- Lock variable can be cached multicore
  - Cache coherence provide the correct Global state of LV
  - LL/SC

Atomic Sync Instructions

- Test and Set (TAS)
- Compare and Swap (CAS)
- Exchange (XCHG)
- Fetch and Increment (FAI)
- How to provide these
  - Load Locked and Store Conditional

test_and_set Instruction : TAS

```
//Definition:
bool test_and_set(bool *LVptr){
    bool rv = *LVptr;
    *LVptr = TRUE;
    return rv:
}
```

- Executed atomically
- Returns the original value of passed parameter
- Set the new value of passed parameter to "TRUE".

CS : test_and_set Instruction : TAS

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
        while (test_and_set(&lock))
        ; /* do nothing */

            /* critical section */

        lock = false;
             /* remainder section */
   } while (true);
```

CAS: Compare and Set

```
int CAS(int *value, int expected, int
                        new_value){
    int temp = *value;
    if (*value == expected)
    *value = new_value;
    return temp;
}
```

- Executed atomically
- Returns the original value of passed parameter “value”
- Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Sync Solution using CAS

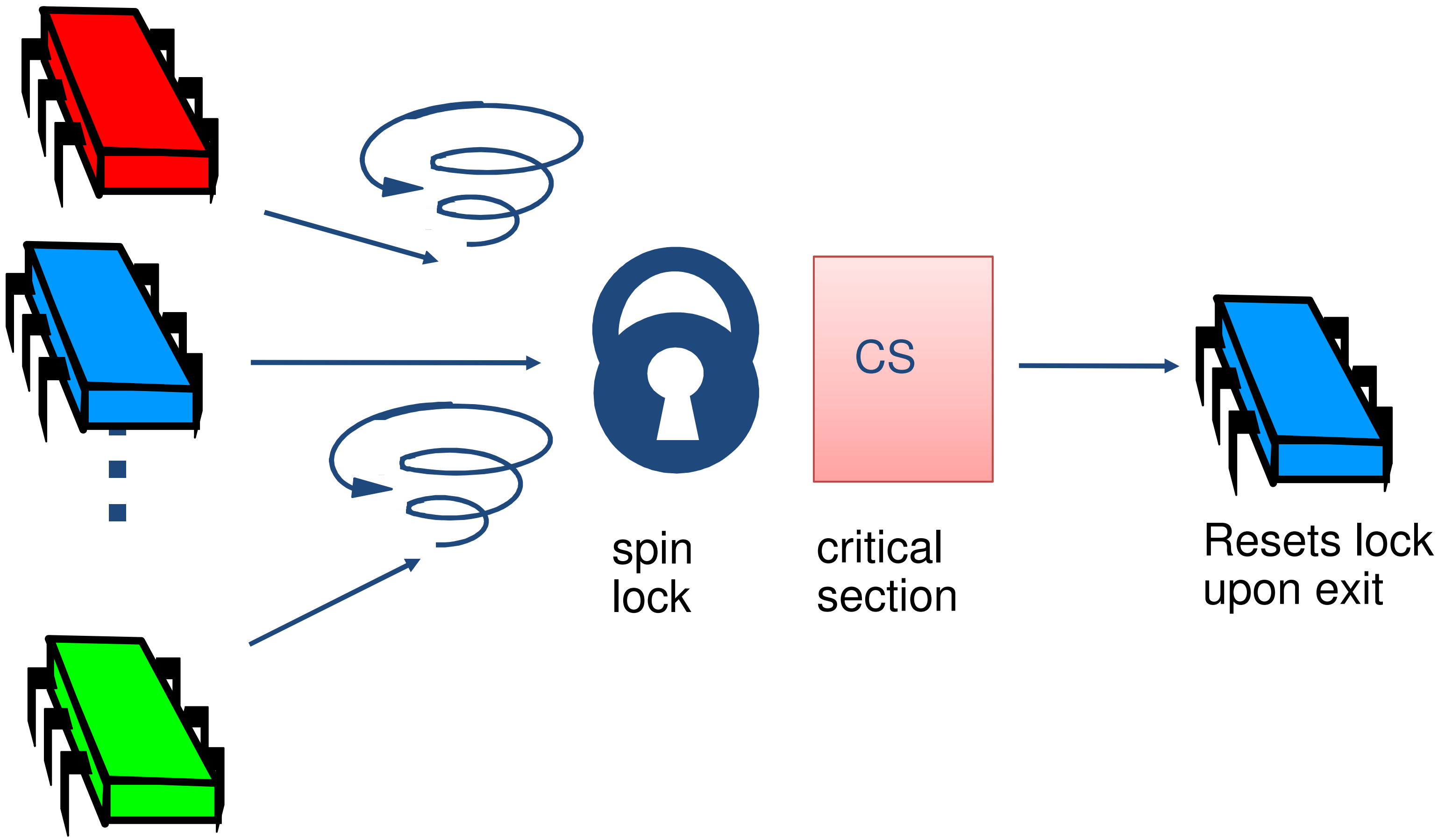
- Shared integer “lock” initialized to 0;

```
do {
  while (CAS(&lock, 0, 1) != 0)
  ; /* do nothing */

   /* critical section */
        lock = 0;
      /* remainder section */

       } while (true);
```

Synchronization Hierarchy ☺ ☺ ☺

- One == (used by)== > other
- **LL+SC** ==> **TAS/CAS/FAI/XCHG**==>**Lock/Unlock**
  - All **TAS/CAS/GAS/FAI/XCHG** do the same work
- Lock/Unlock == > Mutex //Mutex use L/UL
- Mutex == > Semaphore // Semaphore uses Mutex
  - Wait() and Signal()
- Semaphore == > Monitor //Monitor uses Semaphore
  - Many wait/Many Signal, Processes in Queue
  - Monitor : Another Abstract Type
    - *which use semaphore, mutex, conditions*

Many threads trying to acquire Mutex LOCK



Synchronization Primitives

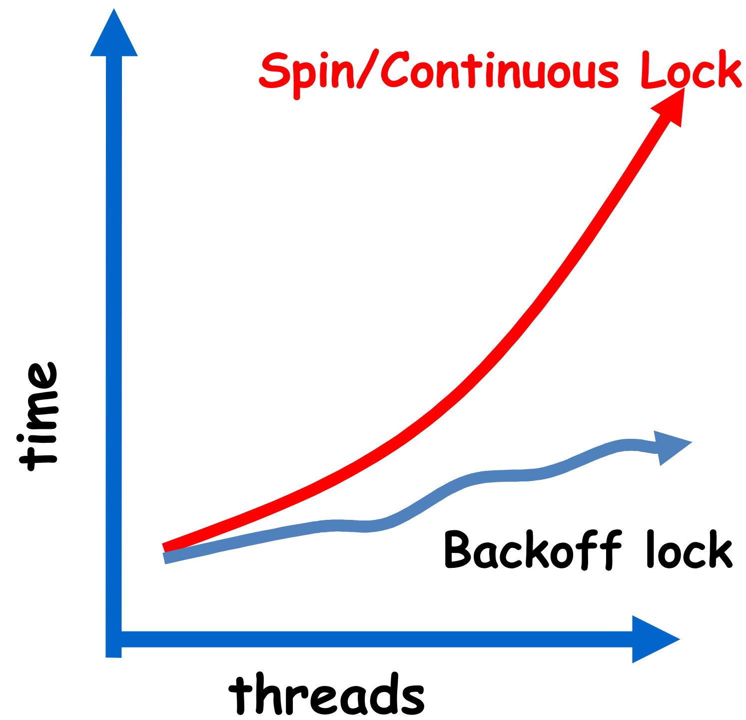
- **int** pthread_mutex_**init**(
  pthread_mutex_t *mutex_lock,
  const pthread_mutexattr_t *lock_attr);
- **int** pthread_mutex_**lock**(
  pthread_mutex_t *mutex_lock);
- **int** pthread_mutex_**unlock**(
  pthread_mutex_t *mutex_lock);
- **int** pthread_mutex_**trylock**(
  pthread_mutex_t *mutex_lock);

Locking Overhead

- Serialization points
  - Minimize the size of critical sections
  - Be careful
- Rather than wait, check if lock is available
  - **pthread_mutex_trylock**
  - If already locked, will return EBUSY
  - Will require restructuring of code
    - **Suspend self by pthread_yeild() Give chance to others**
    - **Suspend self by doing a timed wait..**
      - **{1, 1, 1,..}, {1, 2, 3, 4,..}, {1,2,4,8,..},…**

Performance of Locking

- **Spinning/busy wait waste time**
- Recall MAC Protocol
  - Non Persistence CSMA protocol
  - Wait random time if medium if busy, then send
- Spin lock with exponential back-off reduces contention
  - Wait k amount of time for 1st attempt
  - Wait $k*c^i$ amount of time for i^{th} attempt



Example: simple backup

```
void ExpBackup(int _ival){
int i=0,ival=_ival;
while (i<100) {
  if (m.try_lock()) { ival=_ival;
   BigNonSharedWork();i++;m.unlock();
  } else { ival=ival*2;
   chrono::milliseconds interval(ival);
   this_thread::sleep_for(interval);
   }//End Else
  }//End while
} //EndExpBackup
```

Example :simple backup

```
int main(int argc, char *argv[]){
  thread th[10];
  for(int i=0;i<10;i++)
     th[i]=thread(ExpBackup,
                       atoi(argv[1]));
  for(int i=0;i<10;i++)
     th[i].join();
   return 0;
}
```

Synch. Primitives

- pthread_mutex_init, lock, unlock, trylock
- pthread_attr_setdetachstate, guardsize_np, stacksize, inheritsched, schedpolicy, schedparam
- pthread_cond_wait, signal, broadcast, init, destroy
- Our main concern
  - Lock, unlock, trylock, condsignal, condbroadcast

Condition Variables for Synchronization

- Condition variable allows a thread
  - To block itself until specified data reaches a predefined state.
- Condition variable
  - Associated with a predicate (P)
  - When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.
- Single condition variable
  - May be associated with more than one predicate
  - Ex: P= X OR Y AND (Z OR K)

C++ Thread:atomic

```
atomic_uint AtomicCount;
void DoCount(){
  int j, timesperthrd;
  timesperthrd=(TIMES/NUM_THREADS);
  for(j=0;j<timesperthrd;j++) AtomicCount++;
}

main(){
 thread T[N_THRDS]; int i;
 for(i=0;i<N_THRDS;i++) T[i]=thread(DoCount);
 for(i=0;i<N_THRDS;i++) T[i].join();
}
```

Improved

```
atomic_uint AtomicCount;
void DoCount(){
   int j, timesperthrd,localcount=0;
   timesperthrd=(TIMES/NUM_THREADS);
   for(j=0;j<timesperthrd;j++)localcount++;
           AtomicCount+=localcount;
}
main(){
     thread T[N_THRDS]; int i;
   for(i=0;i<N_THRDS;i++) T[i]=thread(DoCount);
   for(i=0;i<N_THRDS;i++) T[i].join();
}
```