CS330: Operating Systems

Shared address space and concurrency

Recap: Threads

- Threads share the address space
  - Low context switch overheads
  - Global variables can be accessed from thread functions
  - Dynamically allocated memory can be passed as thread arguments
- Sharing data is convenient to design parallel computation
- Pthread API for multi-threaded programming

Threads sharing the address space

- Threads share the address space
    - Global variables can be accessed from thread functions
- Everything seems to be fine, what is the issue?
- How does OS fit into this discussion?
    - Data parallel processing: Data is partitioned into disjoint sets and assigned to different threads
    - Task parallel processing: Each thread performs a different computation on the same data

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
          counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
          counter++;
}
```

counter++ in assembly

```
mov (counter), R1
Add 1, R1
Mov R1, (counter)
```

Even on a single processor system, scheduling of threads between the above instructions can be problematic!

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}

- Assume that T1 is executing the first iteration
- On context switch, value of R1 is saved onto the PCB
- Thread T2 is scheduled and starts executing the loop

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1 // R1 = 0
T2: Add 1, R1 // R1 = 1
T2 mov R1, (counter) // counter = 1
{switch-out, T1 scheduled, R1 = 1}

- T2 executes all the instructions for one iteration of the loop, saves 1 to counter (in memory) and then, scheduled out
- T1 is switched-in, R1 value (=1) loaded from the PCB

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1 // R1 = 0
T2: Add 1, R1 // R1 = 1
T2 mov R1, (counter) // counter = 1
{switch-out, T1 scheduled, R1 = 1}
T1: mov R1, (counter) // counter = 1!

- T1 stores one into counter
- Value of counter should have been two
- What if "counter++" is compiled into a single instruction, e.g., "inc (counter)" ?

Sharing can be problematic!

T1: mov (counter), R1 // R1 = 0
T1: Add 1, R1
{switch-out, R1=1 saved in PCB}
T2: mov (counter), R1 // R1 = 0
T2: Add 1, R1 // R1 = 1
T2 mov R1, (counter) // counter = 1
{switch-out, T1 scheduled, R1 = 1}
T1: mov R1, (counter) // counter = 1!

- T1 stores one into counter
- Value of counter should have been two
- What if "counter++" is compiled into a single instruction, e.g., "inc (counter)" ?
- Does not solve the issue on multi-processor systems!

Sharing can be problematic!

```
static int counter = 0;
void *thfunc(void *)
{
   int ctr = 0;
   for(ctr=0; ctr<100000; ++ctr)
         counter++;
}
```

- If this function is executed by two threads, what will be the value of *counter* when two threads complete?
- Non-deterministic output
- Why?
- Accessing shared variable in a concurrent manner results in incorrect output

Definitions

- Atomic operation: An operation is atomic if it is *uninterruptible* and *indivisible*
- Critical section: A section of code accessing one or more shared resource(s), mostly shared memory location(s)
- Mutual exclusion: Technique to allow exactly one execution entity to execute the critical section
- Lock: A mechanism used to orchestrate entry into critical section
- Race condition: Occurs when multiple threads are allowed to enter the critical section

Threads sharing the address space

- Threads share the address space
  - Global variables can be accessed from thread functions
- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?
    assigned to different threads
  - Task parallel processing: Each thread performs a different computation on the same data

Critical sections in OS

- OS maintains shared information which can be accessed from different OS mode execution (e.g., system call handlers, interrupt handlers etc.)
- Example (1): Same page table entry being updated concurrently because of swapping (triggered because of low memory) and change of protection flags (because of mprotect( ) system call)
- Example (2): The queue of network packets being updated concurrently to deliver the packets to a process and receive incoming packets from the network device

Strategy to handle race conditions in OS

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

Threads sharing the address space

- Threads share the address space
- Everything seems to be fine, what is the issue?
- Correctness of program impacted because of concurrent access to the shared data causes race condition
- How does OS fit into this discussion?
- Concurrency issues in OS is challenging as finding the race condition itself is non-trivial

on the same data

Locking in pthread: pthread mutex

```
pthread_mutex _t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
void *thfunc(void *)
{
  int ctr = 0;
  for(ctr=0; ctr<100000; ++ctr){
    pthread_mutex_lock(&lock);      // One thread acquires lock, others wait
    counter++;                      // Critical section
    pthread_mutex_unlock(&lock);    // Release the lock
  }
}
```

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

```
    pthread_mutex_lock(&lock);      // One thread acquires lock, others wait
    counter++;                      // Critical section
    pthread_mutex_unlock(&lock);    // Release the lock
  }
}
```

Lock ADT

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

```
lock_t *L1, L2;
....
lock(L1)
Critical Section
unlock(L1)
....
lock(L2)
Critical Section
unlock(L2)
....
Lock(L1)
Critical Section
unlock(L2)
```

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations

Lock ADT: Efficiency

```
lock_t *L;

lock(L)
{
  // Return ⇒ Lock acquired
}
unlock(L)
{
  // Return ⇒ Lock released
}
```

- Efficiency of lock/unlock operations directly influence performance
- Implementation choices?
- Hardware assisted implementations
  - Use hardware synchronization primitives like atomic operations
- Software locks are implemented without assuming any hardware support
  - Not used in practice because of high overheads

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Fairness of the locking scheme

```
    counter++;                        // Critical section
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```

Lock: busy-wait (spinlock) vs. Waiting

T1	**T2**
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section
	unlock(L)

Lock: busy-wait (spinlock) vs. Waiting

T1	**T2**
lock(L) //Acquired	
Critical section	lock(L) //Lock is busy. Reschedule or Spin?
unlock(L)	Critical section
	unlock(L)

- With busy waiting, context switch overheads saved, wasted CPU cycles due to spinning
- Busy waiting is prefered when critical section is small and the context executing the critical section is not rescheduled (e.g., due to I/O wait)

Design issues of locks

```
pthread_mutex_t lock;     // Initialized using pthread_mutex_init
static int counter = 0;
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme

```
  }
}
```

Fairness

- Given *N* threads contending for the lock, number of unsuccessful attempts for lock acquisition for all contending threads should be same

Fairness

- Given *N* threads contending for the lock, number of unsuccessful attempts for lock acquisition for all contending threads should be same
- Bounded wait property
  - Given *N* threads contending for the lock, there should be an upper bound on the number of attempts made by a given context to acquire the lock

Design issues of locks

```
pthread_mutex_t lock;    // Initialized using pthread_mutex_init
```

- Efficiency of lock and unlock operations
- Hardware-assisted lock implementations are used for efficiency
- Lock acquisition delay vs. wasted CPU cycles
- Use waiting locks and spinlocks depending on the requirement
- Fairness of the locking scheme
- Contending threads should not starve for the lock indefinitely

```
    pthread_mutex_unlock(&lock);  // Release the lock
  }
}
```