CS330: Operating Systems

Shared address space and concurrency

Threads sharing the address space is useful

- Threads share the address space
    - 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

Threads sharing the address space is useful

- Threads share the address space
  - 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
- Example parallel computation models
  - 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

Example: Finding MAX

- Given *N* elements and a function *f*, we are required to find the element *e* such that *f(e)* is maximum
- If the computation time for function *f* is significant, we can employ multithreading with K threads using the following strategy
- Partition N elements into K non-overlapping sets and assign each thread to compute the MAX within its own set
- When all threads complete, we find out the global maximum

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