Parallel Computer Architecture
Hardware Support for Mutual Exclusion

Instructor: Prof. Naga Kandasamy
ECE Department, Drexel University

April 10, 2019

We will discuss major hardware-based approaches to achieving mutual exclusion.

Interrupt Disabling

In a uniprocessor machine, concurrent processes cannot be overlapped, that is, no two processes can run concurrently. Processes can only be interleaved. Moreover, a process will execute until its time quanta runs out or it is interrupted from the CPU. So, to guarantee mutual exclusion, one can simply prevent a process from being preempted from the CPU by disabling all interrupts. A process can enforce mutual exclusion in the following way.

```
while (1) {
   /* Disable interrupts */

   /* Critical section */

   /* Enable interrupts */

   /* Remainder of code */
}
```

The above approach will not work in a multiprocessor architecture, since if we have more than one processor, multiple processes may be executing at the same time. In this case, disabling the interrupts on one processor does not guarantee mutual exclusion.

Instruction Set Architecture (ISA) Support

Modern processor architectures provide *atomic* instructions as part of the ISA to support mutual exclusion in which an instruction, once started, cannot be interrupted. We will now discuss three commonly implemented instructions.

Atomic Compare and Swap

The definition of the atomic compare and swap function typically is as follows:

```
int atomicCAS (int *mutex, int compareVal, int newVal)
{
   int oldVal = *mutex;

   if (*mutex == compareVal)
      *mutex = newVal;

   return oldVal;
}
```

The above function takes as input the pointer to the mutex variable and compares the value stored at that location—call it `oldVal`—with `compareVal`. If the comparison succeeds, the mutex's value is replaced with `newVal`. The function returns `oldVal` back to the caller. Assume that `mutex` is a shared variable between multiple processes, which is initialized to 0. Each process in the system can use `atomicCAS` to achieve mutual exclusion.

```
int mutex = 0;

while (1) {
   while (atomicCAS(&mutex, 0, 1) != 0);

   /* Critical Section */

   mutex = 0;    /* Reset the mutex */

   /* Remainder of the code */
}
```

Test and Set Instruction

The functionality of the test and set instruction is defined as follows.

```
int testSet (int *mutex)
{
   if (*mutex == 0) {
      *mutex = 1;
      return 1;
   }

   return 0;
}
```

The instruction tests the value of `mutex` which is initialized to 0. If the value is 0, then the instruction sets `mutex` to 1 and returns true. Otherwise, the value is not changed and false is

returned. Again, note that the entire `testSet` function is carried out atomically; if two processes simultaneously attempt to execute the instruction, only one succeeds. The code snippet below shows how mutual exclusion can be implemented by a process within the function `foo` using the `testSet` function. The shared variable `mutex` is first initialized to 0. If multiple processes try to execute the `testSet` instruction, then the only process that may enter the critical section is the one that finds `mutex` equal to 0. The other processes attempting to enter their critical section will go into a busy-waiting (also called spin-waiting) mode, by spinning on the `while` loop. When the process, currently in its critical section, leaves the critical section, it resets `mutex` to 0. Other processes are now free to compete for `mutex`. The winner is the process that happens to execute the `testSet` instruction next.

```
int mutex = 0;     /* This is the shared lock variable */

void foo(void)
{
   while (1) {
      while (testSet(&mutex) == 0);

      /* Critical section */

      mutex = 0;  /* Reset the mutex */

      /* Remainder of the code */
   }
}
```

Exchange Instruction

Like a `testSet`, the `exchange` instruction reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location, it writes whatever value, was in the register to begin with. That is, it atomically exchanges or swaps the value in the memory location and the register. We can implement a lock as before by replacing the `testSet` with an `exchange` instruction as long as we use the values 0 and 1 and ensure that the value in the register is 1 before the swap instruction is executed; the lock has succeeded if the value left in the register by the swap instruction is 0. The functionality of the exchange instruction is defined as follows.

```
void exchange (int *mutex, int registerVal)
{
   int temp;
   temp = *mutex;
   *mutex = registerVal;
   registerVal = temp;
}
```

The following code shows how mutual exclusion is implemented with the `exchange` instruction.

```
int mutex = 0;     /* This is the shared lock variable */

void foo (void)
{
   int key = 1;
   while (1){
      while (key != 0)
         exchange (&mutex, key);

      /* Critical section */

      exchange (&mutex, key);  /* Reset the mutex */

      /* Remainder of the code */
   }
}
```

A shared variable `mutex` is initialized to 0. Each process uses a local variable called `key` that is initialized to 1. The first process that executes `exchange` finds `mutex` equal to 0 and sets `mutex` to 1, thereby preventing any other process from entering the critical section. When it leaves the critical section, it resets `mutex` to 0, allowing another process to gain access to the critical section. The Intel architecture supports an exchange instruction in the ISA called `XCHG`.

Advantages and Disadvantages of Using Special Instructions

Using special machine instructions to enforce mutual exclusion has the following advantages:

- It is applicable to any number of processes on either a single processor or multi-processor systems sharing main memory.
- It can be used to support multiple critical sections, where each critical section can be protected by its own `mutex` variable.

There are some disadvantages:

- When a process is waiting for access to a critical section, it continues to spin on the while loop, and therefore, consumes processor cycles doing nothing.
- When a process exits the critical section, the selection of the next process to enter the critical section is arbitrary, and depends on the scheduler. So, some process could be denied access to the critical section indefinitely.