Chapter 5 Process Synchronization

5.5 Mutex Locks

Operating systems designers build software tools to solve the critical-section problem. The simplest and most basic of these is the *mutex lock*.

```
do {
    acquire(&lock);
        /* critical section */
    release(&lock);
        /* remainder section */
} while (true);
```

A process must `acquire()` the lock before entering a critical section, and then `release()` it at after exiting the section. A process that attempts to acquire an unavailable lock is *blocked* until the lock is released. Thus the mutex lock protect critical regions and prevent race conditions. These operations are defined as:

```
acquire() {                 release() {
  while(!available)             available = true;
       ; /* busy wait */    }
    available = false;
}
```

Calls to either `acquire()` or `release()` must be atomic (usually implemented through `test_and_set()` and `compare_and_swap()`).

The main disadvantage of the implementation is that it requires busy waiting: waiting blocked in a loop for the call to `acquire()`, that is, the CPU is "wasting" work by waiting in the loop. Busy waiting mutexes are also called *spinlock*. Spinlocks have an advantage: they require context switch (which are expensive) when locking a process (compare it to condition variables). Thus, when locks are expected to be held for only short times, spinlocks may be cheaper than other kinds of locks.

5.6 Semaphores

A *semaphore* S is an integer variable accessed with atomic operations `wait()` and `signal()`, defined as follows:

```
wait(S) {                   signal(S) {
    while(S <= 0)               S++;
        ; /* busy wait */    }
    S--;
}
```

Kinds of Semaphores A *counting semaphore* can range over an unrestricted domain. A *binary semaphore* can range between 0 and 1 (thus are equivalent to mutex locks).

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. Init the semaphore to the total available resoureces. When a process wishes to use a resource, performs a `wait(S)` (decrementing the count of available resources). At resource release, perform `signal()` (incrementing the count).

Semaphore Implementation Rather than busy waiting (as in mutex locks), semaphores can be made to block the process when calling a `wait()` operation. Waiting processes are sent to a queue of waiting processes (waiting, in the sense of the operating system process).

When another process calls `signal()`, then a process in the list is restarted by a `wakeup()` operation, which changes the process from waiting state to ready state. A simple implemetation to this definition:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Tip: One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue,

5.7 Monitors

Several complex and tricky errors can ocur when programming with semaphores and locks. To deal with such errors, high-level language constructs called *monitors* are developed.

A *monitor* is an *abstract data type* (ADT) that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor, thus ensuring that only one process at a time is active within the monitor.

Chapter 8 Main Memory

8.1 Background

Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter.

The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore how a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

8.1.1 Basic Hardware.

Main memory and the registers built into the processor itself are the only general-purpose storage that the CPU can access directly. Therefore, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices. If the data are not in memory, they must be moved there before the CPU can operate on them.

For proper system operation we must protect the operating system from access by user processes, and processes from accessing other processes' memory. Since the operating system shouldn't intervene between the CPU and its memory accesses (because of the resulting performance penalty), this protection is implemented at hardware level.

First we need to make sure that each process has a separate memory space. We can provide this protection by using two registers: a *base register*, which holds the smallest legal physical memory addres of the process; and the *limit register*, which specifies the size of the range of memory available for the process (so that all memory addresses `n` that satisfy `base <= n < base + limit`, are available for a given process with such base and limit registers).