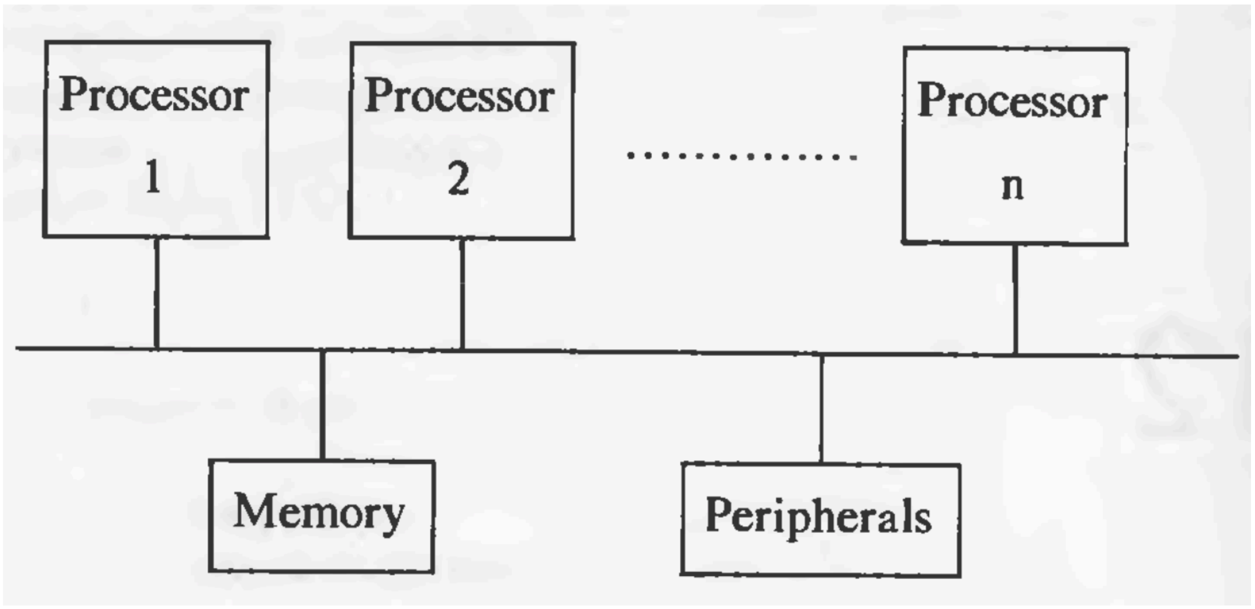
12.Multiprocessor Systems

A multiprocessor architecture contains two or more CPUs that share common memory and peripherals (as shown in the diagram). Processes can run concurrently run on different processors. But each CPU execute the same copy of the kernel. Processes can migrate between processors transparently. But the design of the UNIX system has to be changed to support multiprocessor systems.



Problem of Multiprocessor Systems

As seen previously, the kernel protects the integrity of its data structures by two policies:

- The kernel cannot preempt a process and switch context while executing in kernel mode.
- The kernel masks out interrupts when executing a critical region of code.

These policies are not enough on multiprocessor systems, as two processes might run in kernel mode on independent processors. If that happens, the kernel data structures could easily get corrupt. There are three methods for preventing such corruption:

1. Execute all critical activity on one processor, relying on standard uniprocessor methods for preventing corruption.
2. Serialize access to critical regions of code with locking primitives.
3. Redesign algorithms to avoid contention for data structures.

First two methods are studied here.

Solution with Master and Slave Processors

There could be one processor, the *master*, which can execute in kernel mode and other processors, the *slave*s, execute only in user mode.

The scheduler algorithm decides which processor should execute a process:

```
/*  Algorithm: schedule_process (modified)
 *  Input: none
 *  Output: none
 */

{
        while (no process picked to execute)
        {
                if (running on master processor)
                        for (every process on run queue)
                                pick highest priority process that is loaded in memory;
                else                            // running on a slave processor
                        for (every process on run queue that need not run on master)
                                pick highest priority process that is loaded in memory;
                if (no process eligible to execute)
                        idle the machine;
                        // interrupt takes machine out of idle state
        }
        remove chosen process from run queue;
        switch context to that of chosen process, resume its execution;
}
```

A new field in the process table designates the processor ID that a process must run on. When process on a slave processor makes a system call, the slave kernel sets the processor ID field in the process table, indicating that the process should run only on the master processor, and does a context switch to schedule another process. The algorithm *syscall* is given below:

```
/*  Algorithm: syscall                  // revised algorithm for invocation of system call
 *  Input: system call number
 *  Output: result of system call
 */

{
        if (executing on slave processor)
        {
                set processor ID field in process table entry;
                do context switch;
        }
        do regular algorithm for system call here;
        reset processor ID field to "any" (slave);
        if (other processes must run on master processor)
                do context switch;
}
```

The only chance of corruption is, if two slave processors select the same process to run them. If both processors schedule the same process simultaneously, they would read, write and corrupt its address space.

The system can avoid this in two ways. First, the master can specify the slave processor on which the process should execute, permitting more than one process to be assigned to a processor. Issues of load balancing then arise: One processor may have lots of processes assigned to it, whereas others are idle. The master kernel would have to distribute the process load between the processors. Second, the kernel can allow only once processor to execute the scheduling loop at a time, using mechanisms such as *semaphores*.

Solution with Semaphores

Another method for supporting multiprocessor configuration is to partition the kernel into critical regions such that at most one processor can execute code in a critical region at a time.

Definition of Semaphore

A semaphore is an integer valued object manipulated by the kernel that has the following atomic operations defined for it:

- Initialization of the semaphore to a nonnegative value.
- A *P* (named as *P* by Dijkstra) operation that decrements the value of the semaphore. If the value of the semaphore is less than 0 after decrementing its value, the process that did the *P* goes to sleep.
- A *V* (named as *V* by Dijkstra) operation that increments the value of the semaphore. If the value of the semaphore becomes greater than or equal to 0 as a result, *one* process that had been sleeping as the result of *P* operation wakes up.

- A *conditional P* operation, abbreviated *CP*, that decrements the value of the semaphore and returns an indication of true, if its value is greater than 0. If the value of the semaphore is less than or equal to 0, the value of the semaphore is unchanged and the return value is false.

The semaphores defined here are independent from the user-level semaphores described previously.

Implementation of Semaphores

Dijkstra showed that it is possible to implement semaphores without special machine instructions. C functions to implement semaphores are given below:

```
struct semaphore
{
        int val[NUMPROCS];     // lock --- 1 entry for each processor
        int lastid;            // ID of last processor to get semaphore
};
int procid;              // processor ID, unique per processor
int lastid;              // ID of last proc to get the semaphore

INIT(semaphore)
        struct semaphore semaphore;
{
        int i;
        for (i = 0; i < NUMPROCS; i++)
                semaphore.val[i] = 0;
}
Pprim(semaphore)
        struct semaphore semaphore;
{
        int i, first;

 loop:
        first = lastid;
        semaphore.val[procid] = 1;
 forloop:
        for (i = first; i < NUMPROCS; i++)
        {
                if (i == procid)
                {
                        semaphore.val[i] = 2;
                        for (i = 1; i < NUMPROCS; i++)
                                if (i != procid && semaphore.val[i] == 2)
                                        goto loop;
                        lastid = procid;
```

```
                    return;         // success! now use resource
                }
                else if (semaphore.val[i])
                    goto loop;
        }
        first = 1;
        goto forloop;
}
Vprim(semaphore)
        struct semaphore semaphore;
{
        lastid = (procid + 1) % NUMPROCS;           // reset to next processor
        semaphore.val[procid] = 0;
}
```

TODO : Explain the code above

Most machines have a set of indivisible instructions that do the equivalent locking operation more cheaply, because loops in *Pprim* are slow and would drain performance. For example, the IBM 370 series supports an atomic *compare and swap* instruction, and the AT&T 3B20 computer supports an atomic *read and clear*instruction. When executing the *read and clear* instruction, the machine reads the value of a memory location, clears its value (sets to 0), and sets the condition code according to whether or not the original value was zero. If another processor uses the *read and clear* instruction simultaneously on the same memory location, one processor is guaranteed to read the original value and the other process reads the value 0: The hardware insures atomicity. Thus, the function *Pprim* can be implemented more simply with the *read and clear* instruction:

```
struct semaphore {
        int lock;
};

Init(semaphore)
        struct semaphore semaphore;
{
        semaphore.lock = 1;
}

Pprim(semaphore)
        struct sempahore sempahore;
{
        while (read_and_clear(semaphore.lock))
        ;
}

Vprim(semaphore)
```

```
            struct semaphore semaphore;
{
            semaphore.lock = 1;
}
```

This semaphore primitive cannot be used in the kernel as is, because a process executing it keeps on looping until it succeeds: If a semaphore is being used to lock a data structure, a process should sleep if it finds the semaphore locked, so that the kernel can switch context to another process and do useful work.

Let us define a semaphore to be a structure that consists of a lock field to control access to the semaphore, the value of the semaphore, and a queue of processes sleeping on the semaphore. The value field determines whether a process should have access to the critical region protected by the semaphore.

Algorithm for *P*:

```
/*  Algorithm: P
 *  Input: semaphore
 *        priority
 *  Output: 0 for normal return
 *        -1 for abnormal wakeup due to signals catching in kernel
 *         long jumps for signals not catching in kernel
 */

{
            Pprim(semaphore.lock);
            decrement(semaphore.value);
            if (semaphore.value >= 0)
            {
                        Vprim(semaphore.lock);
                        return (0);
            }
            // must go to sleep
            if (checking signals)
            {
                        if (there is a signal that interrupts sleep)
                        {
                                    increment(semaphore.value);
                                    if (catching signal in kernel)
                                    {
                                                Vprim(semaphore.lock);
                                                return (-1);
                                    }
                                    else
                                    {
                                                Vprim(semaphore.lock);
                        longjmp;
                    }
                }
            }
            enqueue process at end of sleep list of semaphore;
            Vprim(semaphore.lock);
            do context switch;
            check signals, as above;
            return (0);
}
```

At the beginning of the *P* algorithm, the kernel does a *Pprim* operation to ensure exclusive access to the semaphore and then decrements the semaphore value. If the semaphore value is nonnegative, the executing process has access to the critical region: It resets the semaphore lock with the *Vprim* operation so that other processes can access the semaphore and returns an indication of success. If, as a result of decrementing its value, the semaphore value is negative, the kernel puts the process to sleep, following semantics similar to those of regular sleep algorithm: It checks for signals according to the priority value, enqueues the executing process on a first-in-first-out list of sleeping processes, and does a context switch.

The algorithm for *V* is given below:

```
/*  Algorithm: V
 *  Input: address of semaphore
 *  Output: none
 */

{
        Pprim(semaphore.lock);
        increment(semaphore.value);
        if (semaphore.value <= 0)
        {
                remove first process from semaphore sleep list;
                make it ready to run (wake it up);
        }
        Vprim(semaphore.lock);
}
```

Performance Limitations

The increase in system throughput is not linear when more processors are attached to the system. This is because there is more memory contention, meaning that memory accesses take longer. And the contention for semaphores also degrade the performance. In master-slave scheme, the master becomes a bottleneck as the number of processes grows.