Semaphore

The idea: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal.

Example: counter at an elevator.

1. A semaphore can be thought as a non-negative integer, that can only be manipulated by 3 operations.
    a. Init()
    b. semWait()
    c. semSignal()

1. A semaphore may be initialized to a nonnegative integer value.

2. The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.

3. The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

Mutex

A mutex (short for mutual exclusion) is a synchronization primitive used in concurrent programming.

Can be thought of as a lock.

- **Locked**: When a thread acquires the mutex, it is said to be locked. Other threads that try to acquire the mutex will be blocked (or forced to wait) until the mutex is unlocked.
- **Unlocked**: When a thread releases the mutex, it becomes unlocked, allowing other threads to acquire it.
- A mutex typically enforces the rule that only the thread that locked the mutex can unlock it.

Monitor

A software module that consists of one or more procedures, an initialization sequence and local data.

1. The local data variables are accessible only by the monitor's procedures and not by any external procedure.
2. A process enters the monitor by invoking one of its procedures.
3. Only one process may be executing in the monitor at a time; any other processes that have invoked the monitor are blocked, waiting for the monitor to become available.

A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor.

A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:

• cwait (c): Suspend execution of the calling process on condition c. The monitor is now available for use by another process.

• csignal (c): Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

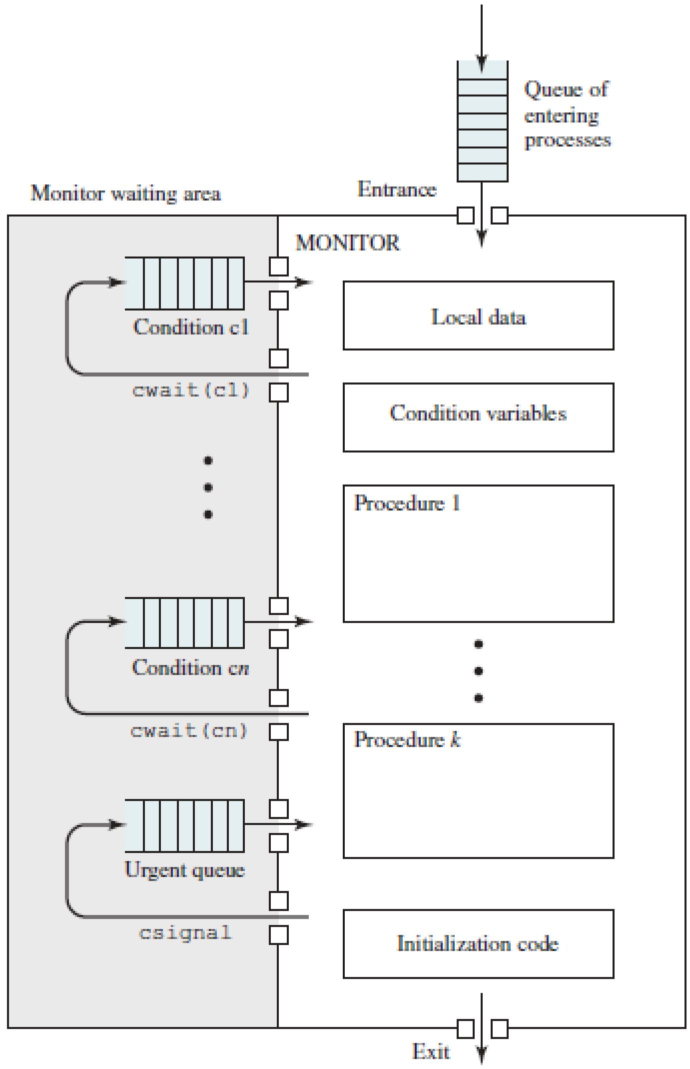

Figure 5.18 Structure of a Monitor

```
monitor dining_controller;
cond ForkReady[5];     /* condition variable for synchronization */
boolean fork[5] = {true}; /* availability status of each fork */

void get_forks(int pid)    /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork[left])
      cwait(ForkReady[left]); /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork[right])
      cwait(ForkReady[right]);/* queue on condition variable */
   fork[right] = false:
   }
void release_forks(int  pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left]) /*no one is waiting for this fork */
      fork[left] = true;
   else               /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])/*no one is waiting for this fork */
      fork[right] = true;
   else               /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]    /* the five philosopher clients */
{
   while  (true) {
     <think>;
    get_forks(k);     /* client requests two forks via monitor */
     <eat spaghetti>;
     release_forks(k); /* client releases forks via the monitor */
   }
}
```

Figure 6.14 **A Solution to the Dining Philosophers Problem Using a Monitor**