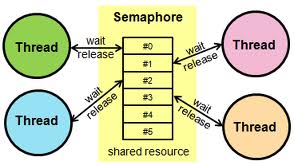
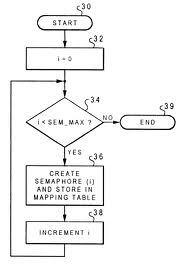
**GETHUB ASSIGNMENT REPORT**

**SEMAPHORE:** Semaphores are devices used to help with synchronization. If multiple processes share a common resource, they need a way to be able to use that resource without disrupting each other. You want each process to be able to read from and write to that resource uninterrupted.

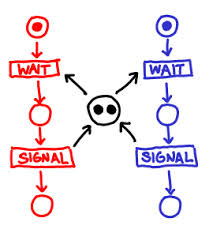


A semaphore will either allow or disallow access to the resource, depending on how it is set up. One example setup would be a semaphore which allowed any number of processes to read from the resource, but only one could ever be in the process of writing to that resource at a time.



**Example of a Semaphore – a Mutex**

A mutex is the most basic type of semaphore, and mutex is short for mutual exclusion. In a mutex, only one thread can use the shared resource at a time. If another thread wants to use the shared resource, it must wait for the owning thread to release the lock.



**MONITOR:** In [concurrent programming](http://en.wikipedia.org/wiki/Concurrent_computing), a **monitor** is an [object](http://en.wikipedia.org/wiki/Object_%28computer_science%29) or [module](http://en.wikipedia.org/wiki/Module_%28programming%29) intended to be used safely by more than one [thread](http://en.wikipedia.org/wiki/Thread_%28computer_science%29). The defining characteristic of a monitor is that its methods are executed with [mutual exclusion](http://en.wikipedia.org/wiki/Mutual_exclusion). That is, at each point in time, at most one thread may be executing any of its [methods](http://en.wikipedia.org/wiki/Method_%28computer_science%29). This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.

Monitors also provide a mechanism for threads to temporarily give up exclusive access, in order to wait for some condition to be met, before regaining exclusive access and resuming their task. Monitors also have a mechanism for signaling other threads that such conditions have been met.

## Differences between Monitors and Semaphores

Both Monitors and Semaphores are used for the same purpose – thread synchronization. But, monitors are simpler to use than semaphores because they handle all of the details of lock acquisition and release. An application using semaphores has to release any locks a thread has acquired when the application terminates – this must be done by the application itself. If the application does not do this, then any other thread that needs the shared resource will not be able to proceed.

Another difference when using semaphores is that every routine accessing a shared resource has to explicitly acquire a lock before using the resource. This can be easily forgotten when coding the routines dealing with multithreading. Monitors, unlike semaphores, automatically acquire the necessary locks.

**CRITICAL SECTION:** A **critical section** is a piece of code that accesses a shared resource (either in the form of data structure or a device) that must not be concurrently accessed by more than one thread of execution (which will otherwise lock it from doing other things).

**BUSY WAITING:** Busy waiting is where a process checks repeatedly for a condition- it is "waiting" for the condition, but it is "busy" checking for it. This will make the process eat CPU (usually).

For example, I have a process that wants to know if there is an internet connection. Here is the psudeocode.

function stay\_running\_until\_there\_is\_internet() {

while (check\_internet() == 0) {\\ wait}}

**SPINLOCK:**

• Suppose a task is running and a little time is left for its completion.

• The running time left for it is less compared to the time that would be taken

in blocking it and context switching.

• There is an innovative concept of spin locking in certain schedulers.

• A ***spin lock*** is a powerful tool in the situation described above.

• The scheduler locking processor for a task waits to cause the blocking of the running

Task first for a time-interval t, then for (t -δt), then (t - 2δt) and so on.

• When this time interval spin downs to 0, the task that requested the lock of the

processor now unlocks the running task and blocks it from further running. The

request is now granted.

• A spin lock does not let a running task be blocked instantly

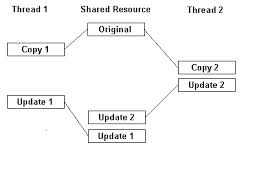
• First successively tries decreasing the trial periods before finally blocking a task

**MUTUAL EXCLSION:** Mutual Exclusion is the concept of restricting access to a shared resource. When multiple processes perform operations on a single resource then they might corrupt it. Its the operating systems' responsibility to make sure that this does not happen. There are many methods that can be used to implement mutual exclusion such as semaphores, monitors, etc. Mutual exclusion has the following properties.

* Safety: No two processes must use the shared resource at the same time. (Should not be in the critical section at the same time.)

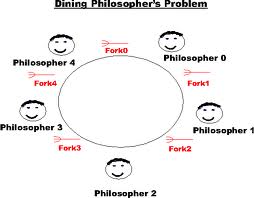


* Liveliness: There should not be deadlocks and a process comes out of the critical section after some time.



* Fairness: A process wanting to use critical section must only wait some time.

**DINING PHILOSOPHER PROBLEM**

****

The Dining Philosophers problem is a classic OS problem that’s usuallu stated in very non-OS terms:

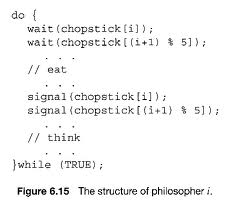
There are *N* philosphers sitting around a circular table eating spaghetti and discussing philosphy.

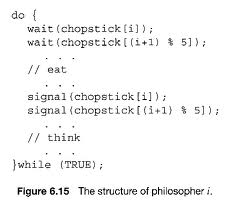
The problem is that each philosopher needs 2 forks to eat, and there are only *N* forks, one

between each 2 philosophers. Design an algorithm that the philosophers can follow that

insures that none starves as long as each philosopher eventually stops eating, and such that the

maximum number of philosophers can eat at once.

****

****