Monitors(one can execute in procedure) has wait queue**:**
pro **con**
share data structure
procedures operate on the shared DS
Sync between concurrent threads that invoke procedures
Monitor invariant-safety property that holds whenever a thread enters or exits the monitor
Condition var-is a condition for a thread to make progress once it is in the monitor
wait-release monitor lock, wait C/V to be signaled
signal-wakeup one waiting thread
Broadcast - wakeup all waiting threads NOT boolean obj
Sinal semantics —> Mesa monitors(java) easier to use, more efficient

Semaphores(binary Mutex with counting, Lock) is abstract data type(integers)
wait(): open -> thread continues l closed -> thread blocks on queue
signal() has “history”(counter), thread on queue, unblocked.
Bounded Buffer(producer, consumer threads)
-Producer inserts resources into the buffer set (output, disk block, memory pages)
-Consumer removes resources from the buffer set (whatever gen by producer)
CPU(%cpu), job throughput(#jobs/time), Turnaround time(Tdone-Tstart),
waiting time(Avg(Twait)), avg time spent on wait queue
Response time Avg(Tready) avg time on ready queue.

Drawbacks: share global vars can be access anywhere in program. used both mutex and scheduling. No control or guarantee of proper usage.

singal() place a waiter on the ready queue, but signaler continues inside monitor
(nr>=0) n (0<=nw <=1)) n (nr>0) -> (nw=0), canWrite(nr=0)n(nw=0)
Multiprogramming-increase CPU utilization job(process,thread) overlapping I/O and CPU activites.
Scheduling Goal-long/short term scheduling happens
long - infrequently significant overhead in swapping a process out to disk
short - scheduling happens relatively frequently (fast CS, fast queue manipulation)
preemptive-systems the scheduler can interrupt a running job(involuntary CS)
non-preemptive-systems, the scheduler waits for a running job to explicitly block(voluntary CS)

Starvation: “non-goal” is a situation where a process is prevented from making progress bc some other process has the resource it requires.(res could be CPU, lock, reader/writer)
side effect of synch-Constant supply of readers always blocks out writers
schduler.alg-Ahigh priority process always prevents a low priority process from running CPU
FCFS/FIFO issue: small jobs wait behind long jobs.
(SJF) -> (AWT) optimal minimum time, **issue** impossible to know size of CPU burst (p or np)
Priority-sch->chosse next job on priority(p l np) issue: low priority wait indefinitely
solution: Age processes(**inc** priority as function wait time, **dec** CPU consumption)
RR(plnp)**->**each time has slice(quantum) no starvation, issue CS frequent&need fast
MLFQ->Queue priority, jobs on same queue sched RR
Processes dynamically change priority
Increases over time if process blocks before end of quantum
dec over time if process uses entire quantum
Deadlock: processes compete for access to limit resources, incorrectly synchronized
Mutex-at least one resource must be held in a non-sharable mode
Hold-n-wait - there must be one process holding one resource and waiting for another resource
No preemption-Resources cannot be preempted
Circular wait-There must exist a set of processes
if the graph has no cycles, DL cannot exist, otherwise DL may exist.
Dealing with DL: ignore it, prevention, Avoidance(control allocation), Detection and Recovery(dependency)
Avoidance:advance about what resources will be needed by processes, avoids circularities
Issue: Hard to determine all resources needed in advance, good theoretical problemnot as practical to us

```
condition::sleep()
disable
waitQ.waitForAccess(c
ur)
conditionlock.release();
KThread.sleep()
unlock
enable
```

```
Semaphore.P () { // wait
Disable interrupts;
if (value == 0) {
add currentThread to
waitQueue;
KThread.sleep(); //
currentThread
}
value = value – 1;
Enable interrupts;
```

```
speak(word)
lock
while(message != null)
        speakCon.sleep(
message = word
listenCon.sleep()
rtn.sleep()
unlock
```

```
monitor Barrier {
  int called = 0; Condition
barrier;
void Done (int needed) {
    called++;
    if (called == needed)
      called = 0;
      barrier.Broadcast();
     else
      barrier.Wait();
```

```
writer
wait(w_or_r)
Write;
signal(w_or_r)
```

```
condtion::wake()
disable
thread = waitQ.nextThread()
if(thread != null)
        thread.ready()
enable
```

```
Semaphore.V()
Disable interruputs
thread = waitQ.current
if(thread != null)
        thread.ready()
else    value++
```

```
int listen()
int ret;
lock
while(message == null)
        listenCV.sleep()
ret = message.intvalue()
message = null
speakCV.wake()
retCV.wake()
unlock
return ret
```

```
class Barrier {
  int called = 0;
  Lock lock;
  Condition barrier;
void Done (int needed) {
  lock.Acquire();
  called++;
  if (called == needed) {
    called = 0;
    barrier.Broadcast(&lock);
  } else {
    barrier.Wait(&lock);
  lock.Release();
```

```
reader
wait(mutex);//lock
readcount+=1;
if(readcount == 1){
wait(wr);} signal(mutex)
Read;
read-=1; if(rd==0);signal(wr)
```

```
class CountdownEvent {
  int counter;
  bool signalled;
  Lock lock;
  Condition cond;
  CountdownEvent (int count) {
    counter = count;
    if (counter > 0) {
      signalled = false;
    } else {
      signalled = true;
    }
    lock = new Lock ();
    cond = new Condition ();
  }
 void Increment () {
    lock.Acquire ();
    if (signalled == false) {
      counter++;
    } // otherwise do nothing if
already signalled

 void Decrement () {
    lock.Acquire ();
    if (signalled == false) {
      counter--;
      if (counter == 0) {
        signalled = true;
        cond.Broadcast ();
      }
    } // otherwise do nothing if
already signalled
    lock.Release ();
  }

  void Wait () {
    lock.Acquire ();
    if (signalled == false) {
      cond.Wait (&lock);
    } // otherwise do nothing if
already signalled
    lock.Release ();
  }
```