Higher-Level Synchronization Mechanisms

AND-Synchronization
Sequencer and Event Count
Monitor

AND Synchronization

- As we have seen with the Cigarette Smoker's Problem, it is often difficult to coordinate access to resources by the means of simple semaphores.
- Why? What was the problem?
- Basically, we need to test for the availability of multiple resources in one step, i.e., tobacco and matches.
- We can do so with AND-Synch. or parallel semaphores.

We define new semaphore operations, SP(s) and SV(s):

SP(s1,t1,d1;s2,t2,d2;.............;sn,tn,dn):
if s1>=t1 and s2>=t2 and ....and sn>=tn then for all i do si:=si-di else place process on queue associated with the first si<ti, establish program counter to restart the entire test.

SV(s1,d1;s2,d2;..............;sn,dn)
si:=si+di for all i; and wakeup all processes waiting on any of the si.

Strong Reader Preference

Shared Variables: NR is a semaphore initialized to the total number of readers, R; mx is a semaphore initialized to 1.

```
Reader

loop
   SP(NR,1,1);
SP(mx,1,0);
    Perform read;
   SV(NR,1);
 endloop;
```

```
Writer
 loop
   SP(mx,1,1; NR, R, 0)
   Perform write;
   SV(mx,1);
 endloop;
```


Writer Preference

Shared Variables: NW is a semaphore initialized to W, the total number of writers; NR is a semaphore initialized to R, the total number of readers; mx is a semaphore initialized to 1.

Writer	Reader
loop	loop
SP(NW,1,1);	SP(NR,1,1; NW,W,0);
SP(mx,1,1;NR,R,0);	Perform read;
Perform write;	SV(NR,1);
SV(mx,1; NW,1);	
endloop;	endloop;

Sequencer and Eventcounts

- An interesting mechanism for coordinating processes is the Sequencer & Eventcount.
- Consider a bakery or a bank with multiple sales persons/clerks.
- Multiple customers can enter the facility at the same time and must be organized such that they are served in order as soon as service is available.
- This problem as been solved by installing a ticket machine that issues tickets in numerical order.
- As soon as any of the servers become available, a counter that displays a number for the Next Customer to be Served is incremented.
- The same mechanism can be used to coordinate and synchronize processes.
- In OS, the stack or of tags by which customers are ordered is represented by an Eventcount E.
- The machine that issues tickets or tags to the customers is represented as a Sequencer S.

…Sequencer & Eventcount

- Only a single operation is defined on the sequencer S:
  - ticket(S) :: issues a non-negative, increasing, and contiguous sequence of integers.
- The ticket(S) operation corresponds to a newly arriving customer taking a unique numbered tag.
- The eventcount E has 3 associated functions:
  - await(E,v);
  - advance(E);
  - read(E);

A sequence

```
v:= ticket(s);
await(E,v)
```

causes the process (customer) to wait until E reaches v.

await(E,v) suspends the calling process if E < v; otherwise it allows the process to proceed.

We may of course combine the two calls above to

```
await(E, ticket(s));
```


Sequencer & Eventcount

Formally, await(E,v) is defined as:

await(E,v):
if E < v then place the calling process in the queue that is associated with E and invoke the scheduler

advance(E) corresponds to an initiation of service: the eventcount value E is incremented and the next process/customer is admitted for service.

advance(s):
E := E + 1;
Wakeup the process(es) waiting for E's value to reach the current value just obtained;

read(E) provides a means for inspecting the current value of E. This may be useful as a process may want to check how long it may have to wait.

Example: Solving the CS problem

```
E: eventcount /* initialized to 0 */
S: sequencer /* initialized to 0 */

await(E, ticket(S));
  enter CS;
advance(E);
```


Example: Producer / Consumer

```
Shared Variables
 var Pticket, Cticket: sequencer;
     In, Out: eventcount;
     buffer: array[0..N-1] of item;
```

!!! Remember, Pticket, Cticket, In, and Out are initialized to 0 !!!

```
Producer i:
 var t: integer;
 loop
  Create new item;
  t:= ticket(Pticket)
  await(In,t); /* one at a time */
  await(Out, t-N+1);
  buffer[t mod N] = item;
  advance(In);
 endloop;
```

```
Consumer k:
 var u: integer;

 loop
  u := ticket(Cticket);
  await(Out, u); /* one at atime */
  await(In, u+1);
  item := buffer[u mod N];
  advance(Out);
  consume item ;
 endloop
```

Monitor – an ADT

- On of the disadvantages of any of the synchronization mechanisms seen thus far is that they are very low-level and hence prone to programming errors.
- A MONITOR is a structure that contains data and functions.
- In other words, a monitor can be viewed as an Abstract Data Type, which facilitates the synchronization and coordination of access to objects.
- The advantages of the object oriented paradigm apply.

- The following are the characteristics of a monitor:
  - Mutual exclusive access → only one process can be active inside the monitor.
  - Access to any of the encapsulated functions is only possible via the monitor procedure.
  - Use of condition variables (CV) → not really variables (no memory is associated with it).
  - Monitor semantics is based on Dijkstra's Guarded Command, e.g.: [condition] → action

..condition variables (CV)

- A condition variable is a name that is chosen by the programmer to represent a specific state or condition. For example:
  - condition: CS_Open

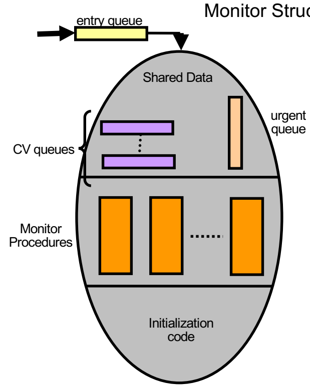
  may be used to represent the fact that the critical section is currently not used.
- Associated with each CV is a queue which can hold processes that are waiting for the condition represented by CV to become "true".

- The following functions are defined for CVs:
  - CV.wait → causes the executing process to be suspended on the queue associated with CV.
  - CV.signal → wakes up a process that waiting on CV, if one exists, otherwise no-op.
  - CV.queue → true if queue is not empty, false if no process is waiting in the queue.
- We need to revisit the semantics after discussing the structure of a monitor.

Queues and more queues

- A monitor maintains many different queues:
  - Entry Queue → sequence processes to ensure mutual exclusive access.
  - CV Queues → store suspended processes that are waiting on the corresponding condition.
  - Urgent Queue → (to implement Hoare semantics) stores processes that have signaled others and are about to exit the monitor.

Monitor Structure



Monitor Examples…

A single resource allocator with Monitors

```
monitor: single_resource
{
  boolean: busy = false; /* initialization */
  condition: non_busy;

  acquire(){
    if busy
      non_busy.wait;
    busy = true;
  }

  release(){
    busy = false;
    non_busy.signal;
  }
}
```

An extension to CV.wait is CV.wait(p) where *p* is a priority number (the smaller *p*, the higher the priority)

Example: Shortest Job First (SJF)

```
monitor: SJN {
  condition x;
  boolean busy = false;

acquire(int: time){
  if busy x.wait(time);
  busy = true;
}

release(){
  busy = false;
  x.signal;
  }
}
```

Different Semantics

- Recall, that only one process can be active inside the monitor at any given moment!
- We need to consider what happens if one process executes a CV.signal, thereby freeing a suspended process.
- This leads to two different CV.signal semantics:
  - Hoare Semantics
  - Mesa Semantics

- In the Hoare semantics
  - the process issuing the CV.signal is placed on the urgent queue if a process was waiting on the corresponding CV-queue.
  - it thereby yields the awakened process (only on process must be active).
  - Upon exiting the monitor, a process signals (V(urgent)) if there are processes suspended in the urgent queue.
  - If the urgent queue is empty, the exiting process will release access to the monitor.

…more semantics…

- In the Mesa Semantics:
  - the process issuing a CV.signal causes a waiting process to be placed on a ready queue.
  - CV.signal actually becomes a notify operation.
  - In this scheme, we cannot guarantee that a particular condition that was signaled to be true, is still true at the time the waiting process will execute.
  - Hence, a waiting process must re-evaluate the condition and suspend itself if it is found false.

·A process can easily re-evaluate the condition by using a while-construct instead of a simple if-construct to test the condition.

while(!B) c.wait;

in1stead of the traditional

if(!B) c.wait;

The Mesa semantics leads to a more efficient implementation and reduces the number of context switches.

Implementing a Monitor

- The following is an implementation of a Monitor ADT by means of semaphores:

```
P(mutex)
//only one process is active in the monitor
…
 body of monitor procedures
…
if next.count > 0   V(next);
 else V(mutex);

// where next is a semaphore that
   represents the urgent queue !!
```

```
CV.wait: →
 CV.count ++;
 if next.count > 0 V(next);
 else V(mutex);
 P(CV_sem);
 CV.count --;

CV.signal: →
 if CV.count > 0 {
   next.count ++;
   V(CV_sem);
   P(next);
   next.count --;
 }
```


Strong Reader Pref. with Monitor

The following is a solution to the strong readers priority using semaphores:

```
monitor: readers_writers;
 int read_count;
 boolean: busy = false
 condition: oktoread, oktowrite;
{

startread() {
   if (busy) oktoread.wait;
   readcount ++;
   oktoread.signal;
 }

endread() {
  readcount --;
  if (readcount == 0) oktowrite.signal;
 }

startwrite() {
  if (readcount > 0 or busy)
     oktowrite.wait;
  busy = true;
}

endwrite() {
  busy = false;
  if (oktoread.queue) oktoread.signal;
  else oktowrite.signal;
}
}
```

… some Monitor exercises

- Try to develop monitor solutions for the following problems:
  - Dining Philosophers
  - Bounded Buffer
  - Cigarette Smokers
- Note: Monitor-based solution usually follow the same style →
  - test and wait on CV are usually the first statements in a monitor procedure.
  - signal or notify are usually the last actions that are performed.