Process Synchronization (IV)

Semaphore Implementation

- Must guarantee:
  - No two processes can actively execute wait( ) on the same semaphore at the same time (but when one is blocked by wait, another is allowed to execute it)
  - No two processes can modify the value of semaphore at the same time (i.e., signal( ) and "S- -" of wait( ) must be atomic)
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.

Semaphore Implementation

All competing processes share a variable lock initialized to false

```
Signal(S) {
        while(testAndSet(&lock));
        S++;
        lock=False;
}
```

Semaphore Implementation

All competing processes share a variable lock initialized to false

```
wait(S) {
                    while(testAndSet(&lock));
                    while (S < =0) {
                              lock=False;
                              while(testAndSet(&lock);
                    }
                    S - -;
                    lock=False;
}
```

Semaphore Implementation

- Ideally, will not have busy waiting in critical section
  - Otherwise, applications may spend lots of time in critical sections.

Semaphore Implementation with no Busy waiting

- With each semaphore, there are
  - value (of type integer)
  - list: an associated waiting queue. Each entry in a waiting queue has two items: (1) pointer to a process/thread; (2) pointer to next record in the list
- Two (atomic) operations:
  - block – place the process invoking the operation on a waiting queue.
  - wakeup – remove one of processes on a waiting queue and place it in the ready queue.

Semaphore Implementation with no Busy waiting

- (Rewritten) Definition of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

Semaphore Implementation with no Busy waiting

- Implementation of wait (example):

```
wait(semaphore *S) {
        while(testAndSet(&lock));
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                lock=False;
                block();
        }
        else lock=False;
}
```

Semaphore Implementation with no Busy waiting

- Definition of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Semaphore Implementation with no Busy waiting

- Implementation of signal (example):

```
signal(semaphore *S) {
        while(testAndSet(&lock));
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
        lock=False;
}
```

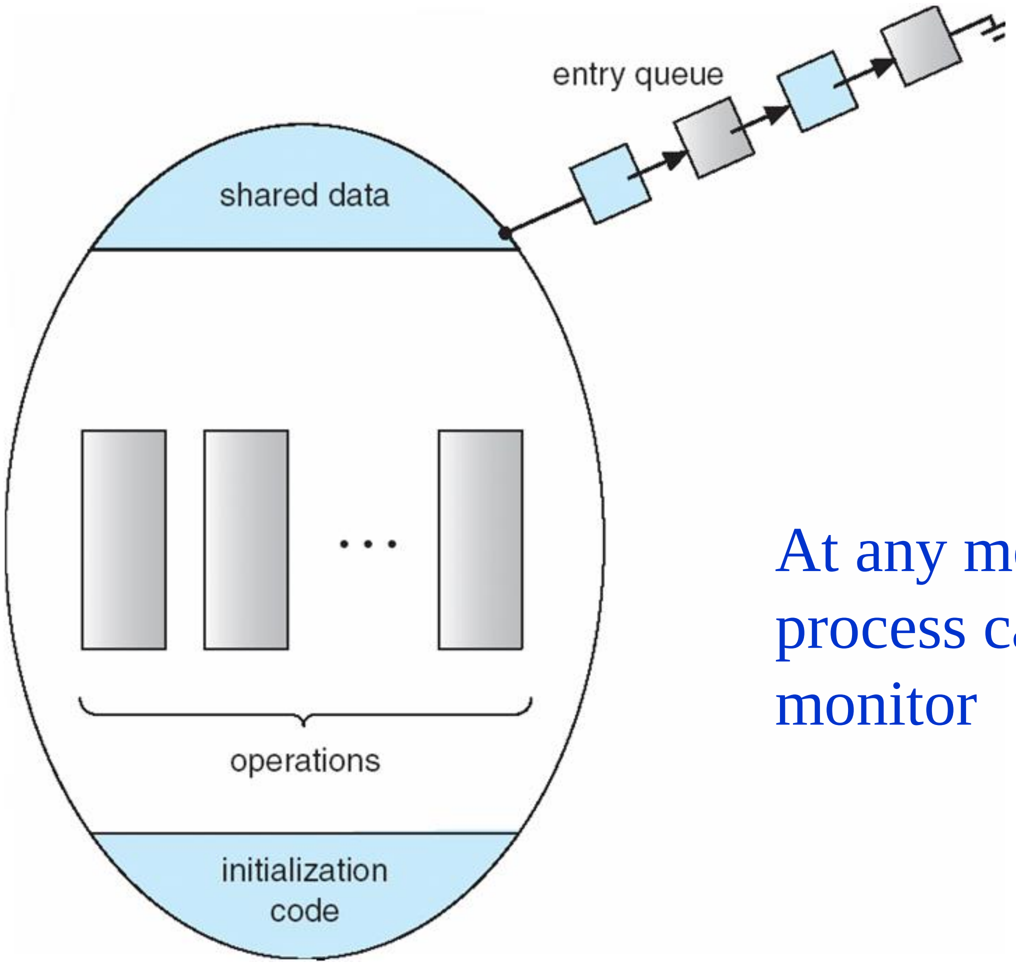
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
          …
  procedure Pn (…) {……}

  Initialization code ( ….) { … }
          …
}
```

Schematic view of a Monitor



At any moment, at most one process can be active in the monitor

Example

type sharedBalance = **monitor**
begin
balance: integer;

procedure credit (amount: integer)
balance := balance +amount
end
procedure debit (amount: integer)
balance := balance - amount;
end

begin /* initialization */
balance := 100;
end
end sharedBalance

sharedBalance.credit(100) **P1**

sharedBalance.debit(100) **P2**

Bounded-Buffer Problem

Step 1: introduce two integers to keep track of the status of the buffer space

- in: how many data items have been ever put into the buffer
- out: how many data items have been ever removed from the buffer

```
producers-consumers: monitor
begin
  in,out: integer;
  pool: 0..9 of buffer;

…
```

```
begin (* initialization *)
  in, out :=0;
end;
```

Bounded-Buffer Problem

Step 2: attempt to compose produce and consume procedures

```
procedure produce (x: buffer);
begin
   while (in >= out+N);
   pool[in mod N] := x;
   in := in+1;
end produce;
```

```
procedure consume(y: buffer);
begin
  while(out >= in);
  y:= buffer[out mod 10];
  out := out+1;
end consume;
```

Bounded-Buffer Problem

Step 2: attempt to compose produce and consume procedures

```
procedure produce (x: buffer);
begin
   while (in >= out+N);
   pool[in mod N] := x;
   in := in+1;
end produce;
```

```
procedure consume(y: buffer);
begin
  while(out >= in);
  y:= buffer[out mod 10];
  out := out+1;
end consume;
```

Wrong! Because the while loop can never end!

Next, we use condition variables to solve this problem …

Condition Variables

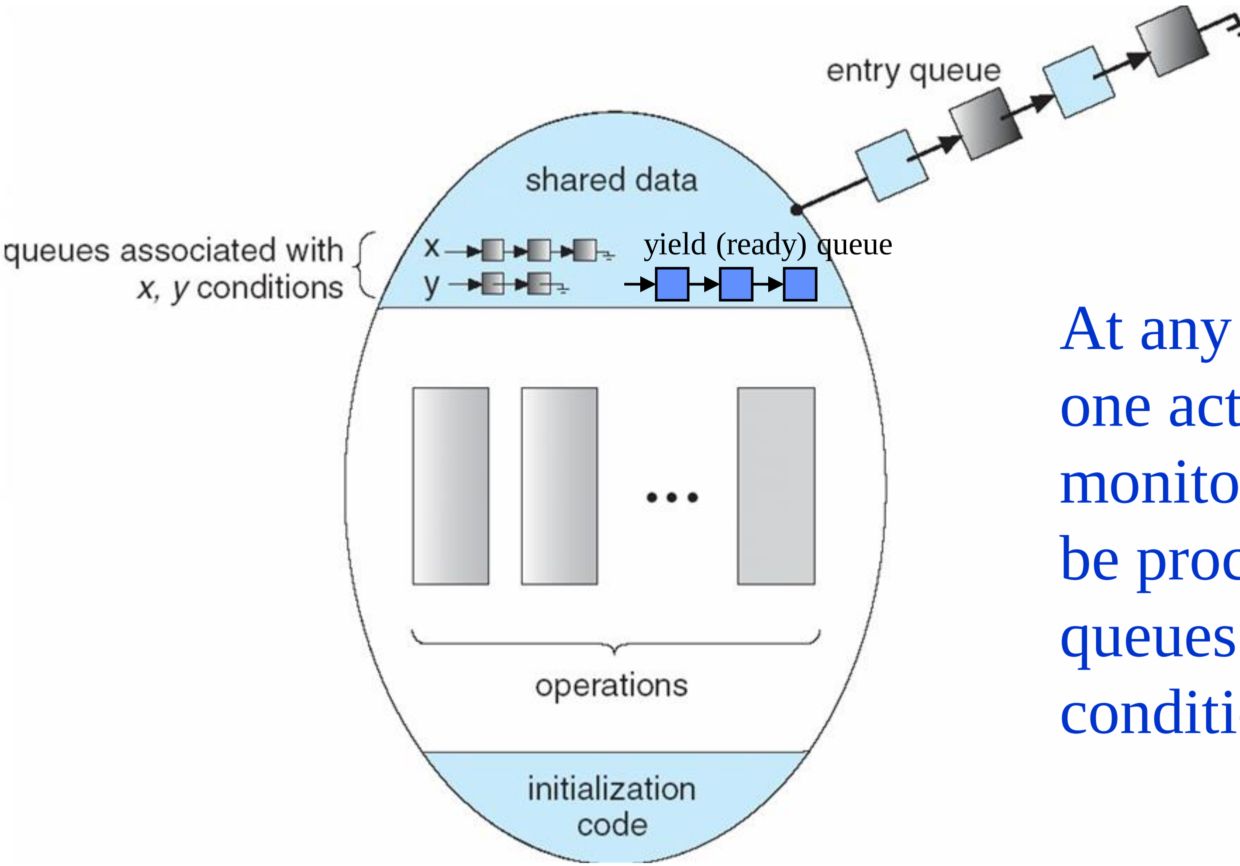
- Monitor can be used to realize mutual exclusion. How to realize more sophisticated synchronization?
- Condition variables
  - Condition variables can be defined in a monitor as defining regular variables

    condition x, y;
  - Two operations on a condition variable:

    x.wait( ) – a process that invokes the operation is suspended.

    x.signal( ) – resumes (wakes up) one of processes (if any) that invoked x.wait (), and blocks the calling process itself unless there is no active process in the monitor (i.e., yield to the woken-up process)

Monitor with Condition Variables



At any moment, at most one active process is in a monitor; but there could be processes blocked in queues associated with conditions or that yield.

Bounded-Buffer Problem

```
producers-consumers: monitor
begin
  in,out: integer
  pool: 0..9 of buffer;
  OKtoproduce, OKtoconsume:
    condition;
procedure produce (x: buffer);
begin
    if in >= out+10 then
     OKtoproduce.wait;
    pool[in mod 10] := x;
    in := in+1;
   OKtoconsume.signal;
end produce;
```

```
procedure consume(y: buffer);
begin
  if  out >= in then
    OKtoconsume.wait;
  y:= buffer[out mod 10];
  out := out+1;
  OKtoproduce.signal;
end consume;

begin (* initialization *)
  in, out :=0;
end;
end producers-consumers;
```