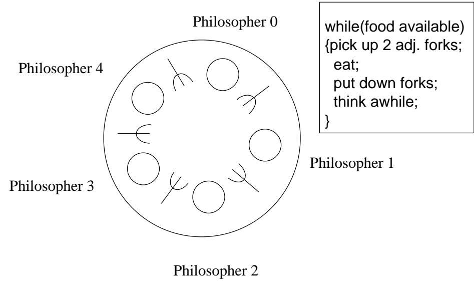
5 Dining Philosophers




```
while(food available)
{pick up 2 adj. forks;
  eat;
  put down forks;
  think awhile;
}
```


Template for Philosopher

```
while (food available)
{   [          ]   /*pick up forks*/
    eat;
    [          ]   /*put down forks*/
    think awhile;
}
```

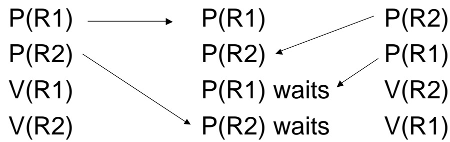

Naive Solution

```
while (food available)
{   P(fork[left(me)]);      /*pick up forks*/
    P(fork[right(me)]);
    eat;
    V(fork[left(me)]);      /*put down forks*/
    V(fork[right(me)]);
    think awhile;
}
```

Does this work?

Simplest Example of Deadlock

Thread 0	Interleaving	Thread 1
P(R1)	P(R1)	P(R2)
P(R2)	P(R2)	P(R1)
V(R1)	P(R1) waits	V(R2)
V(R2)	P(R2) waits	V(R1)



R1 and R2 initially 1 (binary semaphore)

Conditions for Deadlock

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and Thread 1 waits for Thread 0 to V(R1)
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are forcibly removed from their respective holding Threads.

Philosophy 101
(or why 5DP is interesting)

- How to eat with your Fellows without causing ***Deadlock.***
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why ***Starvation*** exists and what we can do about it.

Dealing with Deadlock

It can be ***prevented*** by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- circular waiting
  - Example: Define an ***ordering*** on resources and acquire them in order
- hold and wait
- no pre-emption

Circular Wait Condition

```
while (food available)
{  if (me = =0) {P(fork[left(me)]); P(fork[right(me)]);}
   else {(P(fork[right(me)]); P(fork[left(me)]); }
   eat;
   V(fork[left(me)]); V(fork[right(me)]);
   think awhile;
}
```


Hold and Wait Condition

```
while (food available)
{  P(mutex);
   while (forks [me] != 2)
         {blocking[me] = true; V( mutex); P(sleepy[me]); P( mutex);}
   forks [leftneighbor(me)] --;  forks [rightneighbor(me)]--;
   V(mutex):
   eat;
   P(mutex); forks [leftneighbor(me)] ++;  forks [rightneighbor(me)]++;
   if (blocking[leftneighbor(me)]) {blocking [leftneighbor(me)] = false;
   V(sleepy[leftneighbor(me)]); }
   if (blocking[rightneighbor(me)]) {blocking[rightneighbor(me)] = false;
   V(sleepy[rightneighbor(me)]); }     V( mutex);
   think awhile;
}
```

Starvation

The difference between deadlock and starvation is subtle:

- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.

5DP - Monitor Style

```
Boolean eating [5];
Lock forkMutex;
Condition forksAvail;

void PickupForks (int i) {
    forkMutex.Acquire( );
    while ( eating[(i-1)%5]
    | eating[(i+1)%5] )
        forksAvail.Wait(&forkMutex);
    eating[i] = true;
    forkMutex.Release( );
    }

void PutdownForks (int i) {
    forkMutex.Acquire( );
    eating[i] = false;
    forksAvail.Broadcast(&forkMutex);
    forkMutex.Release( );
    }
```


What about this?

```
while (food available)
{   forkMutex.Acquire( );
    while (forks [me] != 2) {blocking[me]=true;
        forkMutex.Release( ); sleep( ); forkMutex.Acquire( );}
    forks [leftneighbor(me)]--; forks [rightneighbor(me)]--;
    forkMutex.Release( ):
    eat;
    forkMutex.Acquire( );
    forks[leftneighbor(me)] ++; forks [rightneighbor(me)]++;
    if (blocking[leftneighbor(me)] || blocking[ rightneighbor(me)])
        wakeup ( ); forkMutex.Release( );
    think awhile;
}
```

Readers/Writers Problem

Synchronizing access to a file or data record in a database such that any number of threads requesting read-only access are allowed but only one thread requesting write access is allowed, excluding all readers.

102

Template for Readers/Writers

```
Reader()
{while (true)
   {
      /*request r access*/
   read
      /*release r access*/
   }
}
```

```
Writer()
{while (true)
   {
      /*request w access*/
   write
      /*release w access*/
   }
}
```

103

Template for Readers/Writers

```
Reader()
{while (true)
   {
      fd = open(foo, 0);
   read
      close(fd);
   }
}
```

```
Writer()
{while (true)
   {
      fd = open(foo, 1);
   write
      close(fd);
   }
}
```

104

Template for Readers/Writers

```
Reader()
{while (true)
   {
      startRead();
   read
      endRead();
   }
}
```

```
Writer()
{while (true)
   {
      startWrite();
   write
      endWrite();
   }
}
```


R/W - Monitor Style

```
Boolean busy = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead () {
    filesMutex.Acquire( );
    while ( busy )
        OKtoRead.Wait(&filesMutex);
    numReaders++;
    filesMutex.Release( );}

void endRead () {
    filesMutex.Acquire( );
    numReaders--;
    if (numReaders == 0)
      OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );}
```

```
void startWrite() {
    filesMutex.Acquire( );
    while (busy || numReaders != 0)
          OKtoWrite.Wait(&filesMutex);
    busy = true;
    filesMutex.Release( );}

void endWrite() {
    filesMutex.Acquire( );
    busy = false;
    OKtoRead.Broadcast(&filesMutex);
    OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );}
```

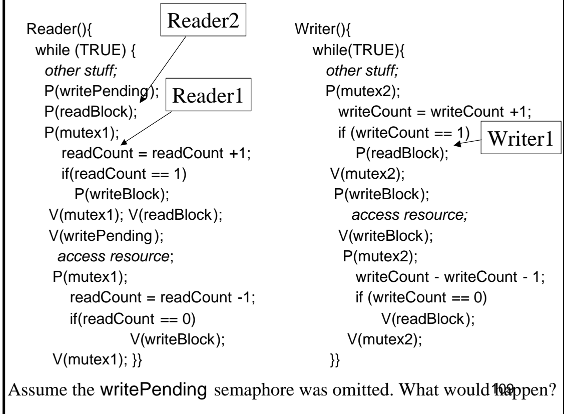

Semaphore Solution with Writer Priority

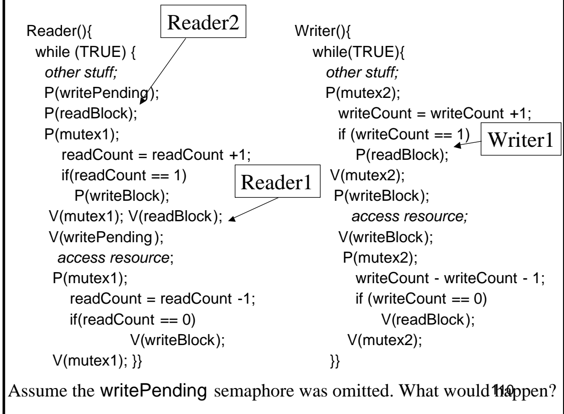
```
int readCount = 0, writeCount = 0;
semaphore mutex1 = 1, mutex2 = 1;
semaphore readBlock = 1;
semaphore writePending = 1;
semaphore writeBlock = 1;
```

```
Reader(){
  while (TRUE) {
    other stuff;
    P(writePending);
    P(readBlock);
    P(mutex1);
      readCount = readCount +1;
      if(readCount == 1)
        P(writeBlock);
    V(mutex1); V(readBlock);
    V(writePending);
    access resource;
    P(mutex1);
      readCount = readCount -1;
      if(readCount == 0)
              V(writeBlock);
    V(mutex1); }}
```

```
Writer(){
  while(TRUE){
    other stuff;
    P(mutex2);
      writeCount = writeCount +1;
      if (writeCount == 1)
        P(readBlock);
    V(mutex2);
    P(writeBlock);
      access resource;
    V(writeBlock);
    P(mutex2);
      writeCount - writeCount - 1;
      if (writeCount == 0)
          V(readBlock);
    V(mutex2);
  }}
```

108





```
Reader(){
  while (TRUE) {
    other stuff;
    P(writePending);
    P(readBlock);
    P(mutex1);
      readCount = readCount +1;
      if(readCount == 1)
        P(writeBlock);
    V(mutex1); V(readBlock);
    V(writePending);
     access resource;
    P(mutex1);
       readCount = readCount -1;
       if(readCount == 0)
              V(writeBlock);
    V(mutex1); }}
```

```
Writer(){
  while(TRUE){
    other stuff;
    P(mutex2);
      writeCount = writeCount +1;
      if (writeCount == 1)
        P(readBlock);
    V(mutex2);
    P(writeBlock);
       access resource;
    V(writeBlock);
     P(mutex2);
        writeCount - writeCount - 1;
        if (writeCount == 0)
            V(readBlock);
      V(mutex2);
  }}
```


Assume the writePending semaphore was omitted. What would happen?

Assume the writePending semaphore was omitted in the solution just given. What would happen?

This is *supposed* to give writers priority. However, consider the following sequence:

Reader 1 arrives, executes through P(readBlock);

Reader 1 executes P(mutex1);

Writer 1 arrives, waits at P(readBlock);

Reader 2 arrives, waits at P(readBlock);

Reader 1 executes V(mutex1); then V(readBlock);

Reader 2 may now proceed…wrong

Practice

Larry, Moe, and Curly are planting seeds. Larry digs the holes. Moe then places a seed in each hole. Curly then fills the hole up.

There are several synchronization constraints:

- Moe cannot plant a seed unless at least one empty hole exists, but Moe does not care how far Larry gets ahead of Moe.
- Curly cannot fill a hole unless at least one hole exists in which Moe has planted a seed, but the hole has not yet been filled. Curly does not care how far Moe gets ahead of Curly.
- Curly *does* care that Larry does not get more than MAX holes ahead of Curly. Thus, if there are MAX unfilled holes, Larry has to wait.
- There is only one shovel with which both Larry and Curly need to dig and fill the holes, respectively.

Sketch out the pseudocode for the 3 threads which represent Larry, Curly, and Moe using whatever synchronization method you like.

Nachos Implementation of Semaphores

```
void
Semaphore::P()
{
   IntStatus oldLevel = interrupt->SetLevel(IntOff);
               // disable interrupts

   while (value == 0) {   // semaphore not available
     queue->Append((void *)currentThread);
     // so go to sleep
     currentThread->Sleep();
   }
   value--;
               // semaphore available,

               // consume its value

   (void) interrupt->SetLevel(oldLevel);            //
     re-enable interrupts
}
```

```
Semaphore::Semaphore(char* debugName, int
     initialValue)
{
   name = debugName;
   value = initialValue;
   queue = new List;
}

void
Semaphore::V()
{
   Thread *thread;
   IntStatus oldLevel = interrupt->SetLevel(IntOff);

   thread = (Thread *)queue->Remove();
   if (thread != NULL)       // make thread ready,
     consuming the V immediately
     scheduler->ReadyToRun(thread);
   value++;
   (void) interrupt->SetLevel(oldLevel);
}
```


Shared and Private Data

- Data shared across threads may need critical section protection when modified
- Data “private” to threads can be used freely
- Private data:
  - Register contents (and register variables)
  - Variables declared local in the procedure fork() starts, and any procedure called by it
  - Elements of shared arrays used by only one thread
- Shared data:
  - Globally declared variables
  - State variables of shared objects