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

Does this work?
**Simplest Example of Deadlock**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Interleaving</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(R1)</td>
<td>P(R1)</td>
<td>P(R2)</td>
</tr>
<tr>
<td>P(R2)</td>
<td>P(R2)</td>
<td>P(R1)</td>
</tr>
<tr>
<td>V(R1)</td>
<td>P(R1) waits</td>
<td>V(R2)</td>
</tr>
<tr>
<td>V(R2)</td>
<td>P(R2) waits</td>
<td>V(R1)</td>
</tr>
</tbody>
</table>

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;
  if (blocking[leftneighbor(me)]) {blocking[leftneighbor(me)] = false; V(sleepy[leftneighbor(me)]); }
  if (blocking[rightneighbor(me)]) {blocking[rightneighbor(me)] = false; V(sleepy[rightneighbor(me)]); }
  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

```c
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?

```c
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();
    fork[leftneighbor (me)]++; fork[rightneighbor (me)]++;
    if (blocking[leftneighbor (me)] || blocking[rightneighbor (me)])
        wakeup / v; 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.

Template for Readers/Writers

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

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

Template for Readers/Writers

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

Writer() {
    while (true) {
        fd = open(foo, 1);
        write
        close(fd);
    }
}
```
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;
Assume the `writePending` semaphore was omitted. What would happen?
Assume the writePending semaphore was omitted. 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

```cpp
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);
}

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
}
```

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