Outline for Today

- Objective:
  - 5 Dining Philosophers
  - Reader-writer problem
  - Message Passing

- Administrative details:
  - Check for demo location with grader
    - TA’s and me: come to our offices
    - UTA’s if they don’t post to newsgroup, ask.

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;
  V(fork[left(me)]);
  V(fork[right(me)]);
  think awhile;
  Does this work?
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)
{
  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(deepy[leftneighbor(me)]);  )
  if (blocking[rightneighbor(me)])(blocking[rightneighbor(me)] = false;  
    V(deepy[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

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();
    }

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

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

Writer()
(while (true))
{
    /*request w access*/
    write
    /*release w access*/
}
### Template for Readers/Writers

**Reader()**

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

**Writer()**

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

**fd = open(foo, 0);**

**fd = open(foo, 1);**

**startRead();**

**endRead();**

**startWrite();**

**endWrite();**

---

### R/W - Monitor Style

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

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

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

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

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

---

### Guidelines for Choosing Lock Granularity

1. **Keep critical sections short.** Push "noncritical" statements outside of critical sections to reduce contention.
2. **Limit lock overhead.** Keep to a minimum the number of times mutexes are acquired and released.
   - Note tradeoff between contention and lock overhead.
3. **Use as few mutexes as possible, but no fewer.**
   - Choose lock scope carefully: if the operations on two different data structures can be separated, it may be more efficient to synchronize those structures with separate locks.
   - Add new locks only as needed to reduce contention.
   - “Correctness first, performance second!”
Semaphore Solution

```c
int readCount = 0;
semaphore mutex = 1;
semaphore writeBlock = 1;
```

Reader()

```c
while (TRUE) {
    other stuff;
P(mutex);
    readCount = readCount +1;
if(readCount == 1)
P(writeBlock);
V(mutex);
access resource;
P(mutex);
readCount = readCount -1;
if(readCount == 0)
V(writeBlock);
V(mutex);
}
```

Writer()

```c
while(TRUE){
    other stuff;
P(writeBlock);
    access resource;
V(writeBlock);
}
```

Attempt at Writer Priority

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

Reader()

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

Writer()

```c
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);
}
```
Reader()
while (TRUE) {
    other stuff;
P(writePending);
P(readBlock);
    readCount = readCount + 1;
    if(readCount == 1)
P(mutex1);
P(writeBlock);
    V(mutex1);
    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(mutex2);
P(writeBlock);
    V(mutex2);
    V(writePending);
    access resource;
P(mutex2);
    writeCount = writeCount - 1;
    if (writeCount == 0)
V(writeBlock);
V(mutex2);}}

Assumptions about semaphore semantics?

Interprocess Communication - Messages

- Assume no explicit sharing of data elements in the address spaces of processes wishing to cooperate/communicate.
- Essence of message-passing is copying (although implementations may avoid actual copies whenever possible).
- Problem-solving with messages - has a feel of more active involvement by participants.
Issues

• System calls for sending and receiving messages with the OS(s) acting as courier.
  – Variations on exact semantics of primitives and in the definition of what comprises a message.
• Naming - direct (to/from pids), indirect (to distinct objects - e.g., mailboxes, ports, sockets)
  – How do unrelated processes “find” each other?
• Buffering - capacity and blocking semantics.
• Guarantees - in-order delivery? no lost messages?

Send and Receive

A common and useful IPC abstraction: Generalized message send and receive primitives.

A messaging interface allows a process to send messages to a particular destination e.g.,

Send and receive are typically system calls, with message queues maintained by the kernel.

5 DP – Direct Send/Receive Message Passing Between Philosophers

Fork please?

Philosopher 0 (thinking)

Philosopher 1

Philosopher 2

Philosopher 3 (eating)

Philosopher 4

Umm. Oh yeah.
Client / Server

One common style of messaging is for a server process to provide services to client processes on demand using request/response message exchanges.

Client

Server
Example: Time Service

A time service could be packaged as a library, using time-related system calls provided by the underlying kernel.

---

Example: Time Service via Messages

The time service may be packaged as a server. Client requests or request time by sending a message to the server and waiting for a response. The client trusts the time server to provide the service correctly, just as they trust the kernel.

---

Client / Server with Threads

- Client blocks until a reply is received.
- Threads allow a client to issue concurrent requests.
- Server waits for a request to arrive.
- Threads allow a server to handle concurrent requests.

---

Hiding Message-Passing: RPC

The request/response communication is a basis for the remote procedure call (RPC) model.

- Think of a server as a module (data + methods).
- Think of a request message as a call to a server method.
- Each request carries an identifier for the desired method; the rest of the message contains the arguments.
- Think of the reply message as a return from a server method.
- Each reply carries an identifier for the matching call; the rest of the message contains the result.

With a little extra glue, the messaging communication can be hidden and made to look “just like a procedure call” to both the client and the server.
Remote Procedure Call - RPC

- Looks like a nice familiar procedure call

P₀

\[ \text{result} = \text{foo}(\text{param}); \]

P₁

Receive

P₀

\[ \text{result} = \text{foo}(\text{param}); \]

P₁

Receive

P₀

\[ \text{result} = \text{foo}(\text{param}); \]

P₁

Receive

P₀

\[ \text{result} = \text{foo}(\text{param}); \]

P₁

Receive

P₀

\[ \text{result} = \text{foo}(\text{param}); \]

P₁

Receive

P₀

\[ \text{result} = \text{foo}(\text{param}); \]

P₁

Receive

P₀

\[ \text{result} = \text{foo}(\text{param}); \]

P₁

Receive
Remote Procedure Call - RPC

• Looks like a nice familiar procedure call

\[ \text{result} = \text{foo}(\text{param}); \]

Receive
\[ r = \text{foo}(\text{param}); \]

// actual call
Reply

Example: Time Service via RPC

5DP via RPC with Fork Manager

• Looks like a nice familiar procedure call

Philosopher\(_0\)

\[ \text{result} = \text{PickupForks}(0); \]

Receive
\[ r = \text{proc}(\text{param}); \]

// explicit queuing
when necessary
Reply
RPC Issues

1. RPC is a syntactically friendly communication/interaction model built above basic messaging or other IPC primitives.
   RPC is a nice model, but it is constrained and not fully transparent; not everyone likes it, and it non-sequentially assumes threads.

2. Complex systems may be structured in the usual way as interacting modules, with processes imposing protection boundaries crossed using RPC.
   Interrelated processes/modules may fail independently (?).

3. The RPC paradigm extends easily to distributed systems, but a variety of optimizations may be employed in the local cases.
   E.g., research systems and X11’s IPC pass arguments to shared memory

4. The RPC model also extends naturally to object-based systems and object-based distributed systems.
   E.g., research systems, CORBA, Java Remote Method Invocation... there is an entire subsection out there