Outline for Today

• Objective:
  – Continue with “classic” concurrency problems
  – Introduce message-passing style interprocess communication
• Administrative details:
  – Handing back problem sets from last week’s discussion sessions...

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.

Template for Philosopher Threads

```java
while (food available) {
    PickupForks (me);
    eat;
    PutdownForks (me);
    think awhile;
}
```

5DP - Monitor Style

```java
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();
} 
```
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);
  think awhile;
}

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();
  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.

Template for Readers/Writers

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

Writer()
{ while (true)
  { write
    *request w access*/
    write
    *release w access*/
  }
}
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);
        OKtoRead.Broadcast(&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();
}

Guidelines for Choosing Lock Granularity

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 with Writer Priority

```c
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 in the solution just given. What would happen?

This is supposed to give writers priority. However, consider the following sequence:
Reader 1 arrives, executes thro’ 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

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.,

\begin{verbatim}
thread->send(data);
currentThread->receive(data);
\end{verbatim}

Like pipes, messaging combines synchronization and data transfer.

Messages for a given destination are stored in a queue pending delivery.

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

---

5 DP – Direct Send/Receive Message Passing Between Philosophers

Philosopher 0

Philosopher 1

Philosopher 2

Philosopher 3

Philosopher 4

Umm. Oh yeah.

Fork please?

Philosopher 0

Philosopher 1

Philosopher 2

Philosopher 3

Philosopher 4

Fork please?
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.

```
Thread* client;
while(systemActive) {
  client->send(request);
  response = currentThread->receive();
  handle the request
}
```

Example: Time Service

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

```
application
|____________________________|
|                            |
|                            |
|____________________________|
```
Example: Time Service via Messages

Client / Server with Threads

Hiding Message-Passing: RPC

Remote Procedure Call - RPC

The time service may be packaged as a server; clients pause 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 blocks until a reply is received.
- Threads allow a client to issue concurrent requests.
2. Server waits for a request to arrive.
- Threads allow a server to handle concurrent requests.

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.

*Looks* like a nice familiar procedure call

\[ \text{result} = \text{foo(param)}; \]
Remote Procedure Call - RPC

- Looks like a nice familiar procedure call

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

Please do \text{foo} for \( P_0 \) with \text{param}

Receive

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

// actual call

Reply

returning \text{r} to \( P_0 \)

Blocked here
Remote Procedure Call - RPC

- Looks like a nice familiar procedure call

\[ \text{P}_0 \]
result = foo(param);

\[ \text{P}_1 \]
Receive r = foo(param);
// actual call
Reply

5DP via RPC with Fork Manager

- Looks like a nice familiar procedure call

\[ \text{Philosopher}_0 \]
result = PickupForks(0);

\[ \text{Fork Server} \]
Receive r = proc(param);
// explicit queuing when necessary
Reply

Example: Time Service via RPC

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-trivially assumes threads.

2. Complex systems may be structured in the usual way as interacting modules, with processes imposing protection boundaries crossed using RPC.

   Interacting processes/machines 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 NT’s IPC pass arguments in 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 subculture out there.
Naming Destinations for Messages: Ports

It may be useful for a given process to manage multiple communication endpoints—often called ports—so that messages sent to ports rather than processes:

Advantages of Ports

1. Ports decouple IPC endpoints from processes and threads. A thread may send to a port without knowing the identity of the process/thread that receives on that port.
2. Different threads may listen on the same port, possibly at different times.
3. A thread may listen on multiple ports, separating the message streams designated for different ports.
4. E.g., assign different ports to different objects or virtual services.
5. Ports are a convenient granularity to control message flow. E.g., selectively enable/disable ports independently, or assign different priorities or access control to different ports.

Port Issues

1. Asynchrony and notification. How does a thread know when a message arrives on a port?
   How to receive from multiple ports without blocking on an idle port while incoming messages are queued on another?
2. Naming and binding. How do threads name the ports to send to or receive from (listen)?
   How do threads find the names, e.g., for services they want to use?
3. Protection and access control.
   How does the system know if a thread/process has a "right" to send to or listen on a particular port? E.g., how can we prevent untrusted programs from masquerading as a legitimate service?

Examples of Ports in Real Systems

1. Unix sockets and TCP/IP communication.
   - Common primitives/protocol for local messaging and network communication.
   - TCP provides a fixed space of port numbers per node.
   - System calls to send/receive from a particular port.
   - Some ports are reserved for processes running with superuser (root) privilege.
   - Standard servers in /etc/services listen on well-known protected ports.
2. Mac supplies a rich set of port/messaging primitives.
   - Open ports (port rights) are kernel object handles.
   - Port rights may be passed in messages among processes.
   - The only way to get a send/receive right is for some other process to pass it to you. This is a system-wide basis for protection.
Notification of Pending Messages

Communication-oriented systems face an important problem:

- How does a client or server know what to do next?
- Servers in networks or server-structured systems might service many clients, possibly on different ports.
- The server must handle messages as they arrive, without blocking to receive on an empty port while others have pending messages.

**Option 1:** Use blocking primitives with lots of threads.
Leave the scheduling to the thread scheduler.

**Option 2:** Introduce nonblocking primitives or provide notifications or combined queueing of incoming messages.
A wide variety of mechanisms have been used: nonblocking polling, Unix select, Mackport groups, event queues, etc.

Polling: Select

A thread/process with multiple network connections or open files can initiate nonblocking I/O on all of them.

The Unix `select` system call supports such a polling model:
- pass a bitmap for which descriptors to query for readiness
- returns a bitmap of descriptors ready for reading/writing
- reads and/or writes on these descriptors will not block

Immediate Notification: Upcalls

Problem: what if an event requires a more “immediate” notification?

- What if a high-priority event occurs while we are executing the handler for a low-priority event?
- What about exceptions relating to the handling of an event?

We need some way to preemptively “break in” to the execution of a thread and notify it of events.

- `upcall`
- example: NT Asynchronous Procedure Calls (APCs)
- example: Unix signals

Preemptive event handling raises synchronization issues similar to interrupt handling.

Advantages of Server “Isolation” Afforded by Message Passing

Like the kernel, the server is protected from its clients.

- Address space isolation is preserved, so the client cannot corrupt the server’s data.
- The only way a client can cause code to run in the server is to send a message.
  The server decides how to validate and interpret each message.
- The client is also protected from the server, although it must rely on it to correctly perform the service.
  (Unlike the kernel, the server cannot access client memory.)

Protected servers may coordinate interactions among processes, manage system-critical data, or otherwise assume roles “typically” reserved for the operating system kernel.
Reconsidering the Kernel Interface and OS Structure

The kernel can be thought of as nothing more than a server; it is special only in that it runs in a protected hardware mode.
- Many of the services traditionally offered by the kernel can be supported outside of the kernel, in servers or in libraries.
- What features must be implemented in the kernel? Could we implement (say) the entire Unix interface as an application?
- Why would we want to do such a thing?
  - What are the advantages of supporting some OS feature in a server rather than directly in the kernel? What are the costs?
  - How would we design a kernel interface that is powerful enough to implement multiple OS “personas” as servers?

The kernel interface is not the programming interface.

Servers and Microkernels

A number of systems have been structured as collections of servers running above a minimal kernel (“microkernel”).
- Microkernel provides, e.g., basic threads and scheduling, IPC, virtual address spaces, and device I/O primitives.
  - Kernel is hoped to be smaller, more reliable, and more secure.
  - Policies (e.g., security) may be implemented outside of the kernel.
- Operating system “personas” (e.g., Unix or Windows) may be implemented as servers.
  - OS may have multiple personalities and policies, with new OS features and APIs added on-the-fly.
- The performance of server-structured systems is determined largely by the efficiency of the messaging primitives.

Microkernel with “User-Level” OS Server Processes

End-to-End Argument

- Application-level correctness requires checking at the endpoints to ensure that the message exchange accomplished its purpose
  - Application semantics involved
  - Notification of successful delivery (UPS tracking) is not as good as a direct response (thank you note) from the other end.
- Reliability guarantees in the message-passing subsystem provide performance benefits (short-circuiting corrective measures).
  - Re-transmitting packet may save retransferring whole file.
Next Time

- Scheduling
- Unix API for Processes