

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...

1

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.

2

Template for Philosopher Threads

```
while (food_available)
{
    PickupForks(me);
    eat;
    PutdownForks(me);
    think awhile;
}
```

3

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( );
}
```

4

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;
}

```

5

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;
}

```

6

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.

7

Template for Readers/Writers

<pre> Reader() {while (true) { /*request r access*/ read /*release r access*/ } } </pre>	<pre> Writer() {while (true) { /*request w access*/ write /*release w access*/ } } </pre>
--	---

8

Template for Readers/Writers

```

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

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

9

Template for Readers/Writers

```

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

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

10

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( );
}
    
```

11

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!"

12

Semaphore Solution with Writer Priority

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

13

```
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);
  }}
```

14

```
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);
  }}

Reader1
Reader2
Writer1
```

Assume the writePending semaphore was omitted. What would happen?

```
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);
  }}

Reader1
Reader2
Writer1
```

Assume the writePending semaphore was omitted. What would happen?

```

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 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**

18

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.

20

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?

21

Send and Receive

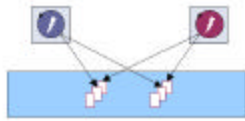
A common and useful IPC abstraction: Generalized message *send* and *receive* primitives.



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

```
thread->send(data);
currentThread->receive(data);
```

Like pipes, messaging combines synchronization and data transfer.

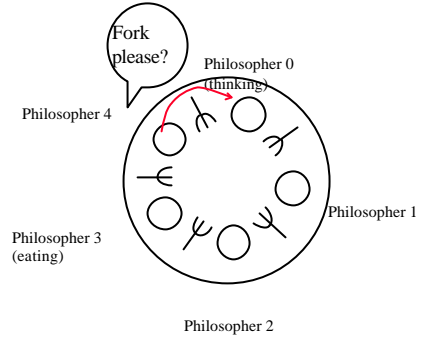


Messages for a given destination are stored in a queue pending delivery.

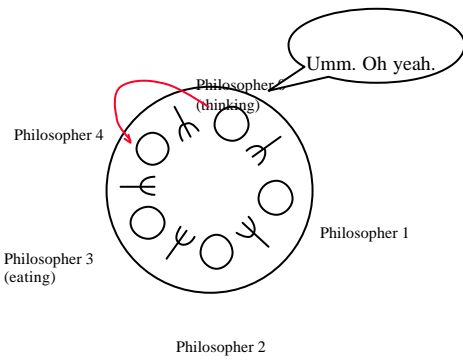
Send and *receive* are typically system calls, with message queues maintained by the kernel.

22

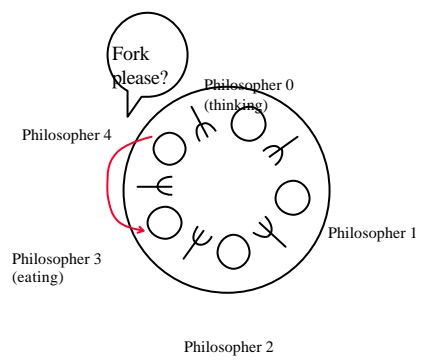
5 DP – Direct Send/Receive Message Passing Between Philosophers



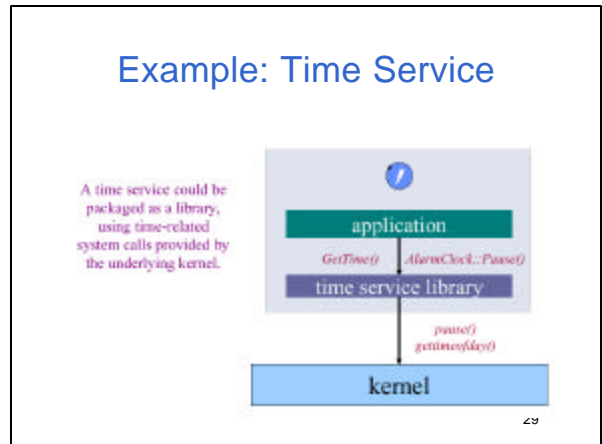
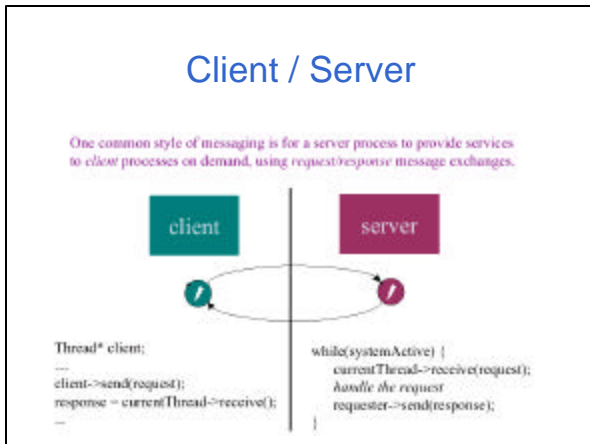
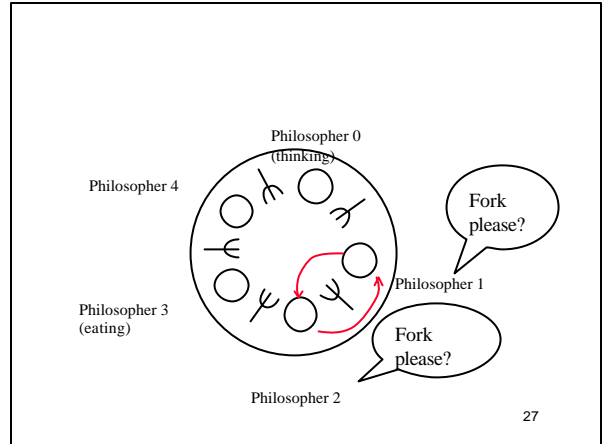
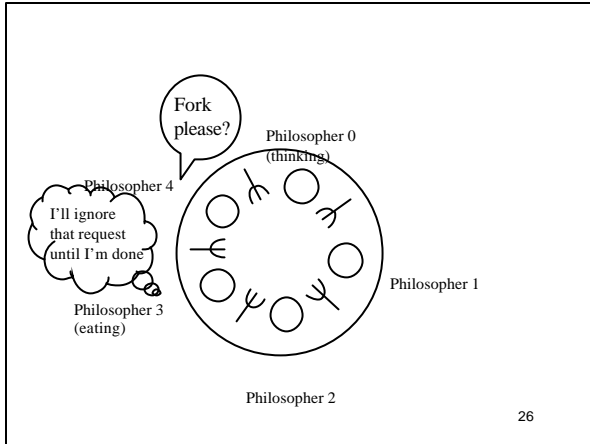
23



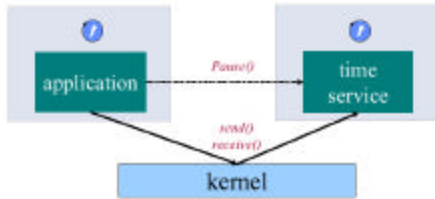
24



25



Example: Time Service via Messages



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 clients trust the time server to provide the service correctly, just as they trust the kernel.

Client / Server with Threads



Now the server with threads:

1. 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.

31

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 glow, the messaging communication can be hidden and made to look "just like a procedure call" to both the client and the server.

32

Remote Procedure Call - RPC

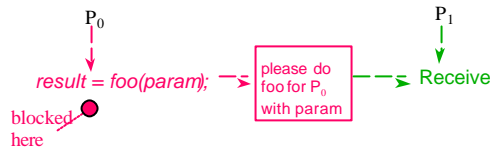
- Looks like a nice familiar procedure call



33

Remote Procedure Call - RPC

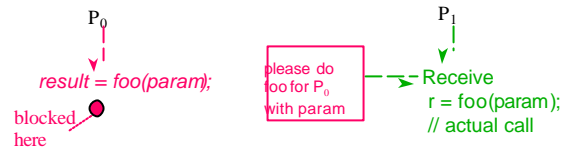
- Looks like a nice familiar procedure call



34

Remote Procedure Call - RPC

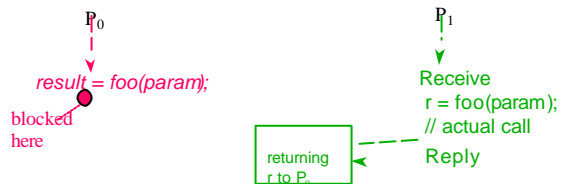
- Looks like a nice familiar procedure call



35

Remote Procedure Call - RPC

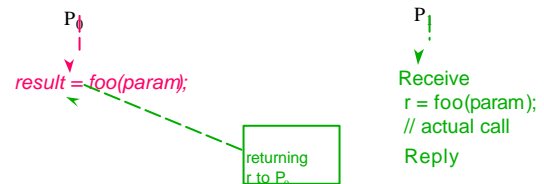
- Looks like a nice familiar procedure call



36

Remote Procedure Call - RPC

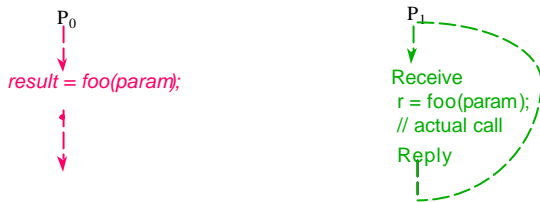
- Looks like a nice familiar procedure call



37

Remote Procedure Call - RPC

- Looks like a nice familiar procedure call



38

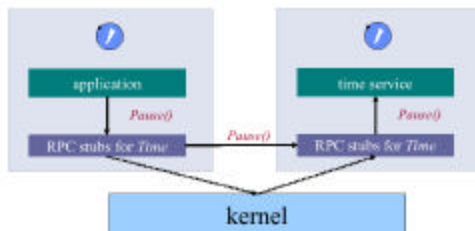
5DP via RPC with Fork Manager

- Looks like a nice familiar procedure call



39

Example: Time Service via RPC



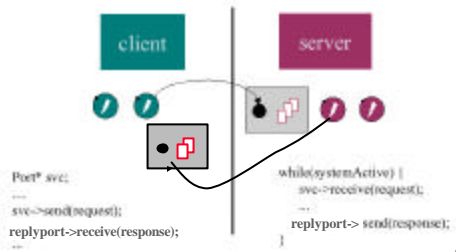
RPC stubs are library routines that handle the details of interacting with the server/client. They may be generated by the system automatically from an abstract description of the service (e.g., a module header file).

RPC Issues

- 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 more-or-less assumes threads.
- Complex systems may be structured in the usual way as interacting modules, with processes imposing protection boundaries crossed using RPC.
 - Interacting processes/modules may fail independently (?).
- 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 *LPC* pass arguments in shared memory
- 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 - with 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.

Different threads may listen/service the same port, possibly at different times.

2. A thread may listen to multiple ports, separating the message streams designated for different ports.

E.g., assign different ports to different objects or virtual services.

3. 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.

44

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?

45

Examples of Ports in Real Systems

1. Unix sockets and TCP/IP communication.

- Common primitives/protocols for local messaging and network communication.

- TCP/IP defines a fixed space of port numbers per node.

System calls to send/listen to a particular port.

- Some ports are reserved to processes running with superuser (root) privilege.

Standard servers in `/etc/services` listen at well-known protected ports.

2. Mach 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.

46

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 queuing of incoming messages.

A wide variety of mechanisms have been used: nonblocking polling, Unix `select`, Mach port groups, event queues, etc.

47

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 bitmask for which descriptors to query for readiness
- returns a bitmask of descriptors ready for reading/writing
- reads and/or writes on these descriptors will not block



Select has fundamental scaling limitations in storing, passing, and traversing the bitmaps.

48

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.

upcalls

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 "personalities" as servers?
 - The kernel interface is not the programming interface!*

52

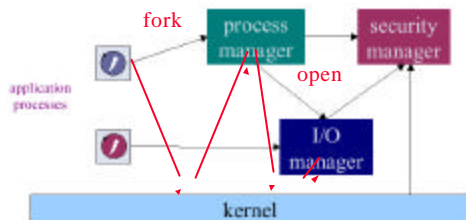
Servers and Microkernels

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 "personalities" (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.

53

Microkernel with "User-Level" OS Server Processes



54

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 re-transferring whole file.

56

Next Time

- Scheduling
- Unix API for Processes