# Concurrent Programming in Java: Creating Threads

Creating threads in Java, including oneway messages, services in threads, and parallel decomposition.

It is impossible to categorize all the ways to exploit the functionality associated with threads. But two general approaches can be distinguished by their points of view on the statement:

new Thread(aRunnable).start();

Is this a fancy way to invoke a method (i.e., a Runnable's run method), or is it a way to create a fancy object (i.e., a new instance of class Thread)? Clearly it is both, but focusing on one aspect versus the other leads to two approaches to using threads that were implicit in discussions in Chapter 1:

**Task-based.** Here, the main reason to use a thread is to asynchronously invoke a method that performs some task. The task might range from a single method to an entire session. Thread-based techniques can support message-passing schemes that escape the limitations of pure procedural calls. Task-based designs are seen in event frameworks, parallel computation, and IO-intensive systems.

**Actor-based.** Here, the main reason to use a thread is to create and set into motion a new autonomous, active, process-like object. This object may in turn react to external events, interact with other actors, and so on. Actor-based designs are seen in reactive, control, and distributed systems. They are also the focus of most formal approaches to concurrency.

(Both the terms *task* and *actor* have many overloaded meanings and near-synonyms. We'll confine usage to the above senses.)

In task-based systems, passive objects sometimes send active (thread-propelled) messages, while in actor-based systems, active objects normally send passive messages. As is usually the case for artificial dichotomies, neither approach is always best, and there is a huge middle ground that can be designed from either or both perspectives.

Actor-based approaches are commonly used in the construction of daemons that interact with other systems. They are also employed when defining intrinsically active entities, for example theGamePlayer in 3.2.4. Their main methods often take a reactive looping form:

for(;;) { acceptAndProcessCommand(); }

Task-based approaches are commonly used when there is some conceptual or performance-based reason to execute a given task, service, or computation asynchronously rather than relying on direct procedural invocation. Task-based designs provide a separation of concerns between logical asynchrony and mappings to threads and thread-based constructions. They receive the bulk of discussion in this chapter.

As an initial example, here is one way to approach a common thread-based design, a web service. Here, a running WebService is a "daemon process" actor-style thread — it continuously interacts with its environment by listening for new incoming requests. But invocations to handler.process are issued in a task-based manner — a new task is set in motion to handle each incoming request. Here, for the sake of concise illustration, the request is simply a number, and the handler just returns the negation of the number back to the client.

[**Figure 4-1**](javascript:popUp('/content/images/0201310090/elementLinks/04fig01.gif'))

class WebService implements Runnable {

static final int PORT = 1040; // just for demo

Handler handler = new Handler();

public void run() {

try {

ServerSocket socket = new ServerSocket(PORT);

for (;;) {

final Socket connection = socket.accept();

new Thread(new Runnable() {

public void run() {

handler.process(connection);

}}).start();

}

}

catch(Exception e) { } // die

}

public static void main(String[ ] args) {

new Thread(new WebService()).start();

}

}

class Handler {

void process(Socket s) {

DataInputStream in = null;

DataOutputStream out = null;

try {

in = new DataInputStream(s.getInputStream());

out = new DataOutputStream(s.getOutputStream());

int request = in.readInt();

int result = -request; // return negation to client

out.writeInt(result);

}

catch(IOException ex) {} // fall through

finally { // clean up

try { if (in != null) in.close(); }

catch (IOException ignore) {}

try { if (out != null) out.close(); }

catch (IOException ignore) {}

try { s.close(); }

catch (IOException ignore) {}

}

}

}

This chapter divides coverage of thread construction and structuring techniques as follows:

* 4.1 presents a series of options for implementing conceptually oneway messages, sometimes by asynchronously initiating tasks using threads or thread-based lightweight execution frameworks.
* 4.2 discusses the design of systems in which networks of components employ oneway messaging strategies.
* 4.3 presents alternatives for constructing threads that compute results or provide services to clients that initiate them.
* 4.4 examines problem decomposition techniques that can be used to improve performance by exploiting multiprocessors.
* 4.5 provides an overview of constructs and frameworks for designing systems of active objects, illustrated in part using CSP.

Many of the designs presented in this chapter straddle the borders among concurrent, distributed, and parallel programming. Presentations focus on concurrent, single-JVM solutions. But they include constructions often seen when developing the plumbing support for systems and frameworks involving multiple processes or computers.

## 4.1 Oneway Messages

A host object issues a logically oneway message to one or more recipients without depending on the consequences of that message. Sending a oneway message somehow results in some task being performed. The task might consist of only a single line of code, or might represent a session that entails acquisition of many resources and hours of computation. But the outcome of the thread issuing a oneway message does not rely on the task's outcome, or on when the task completes, or (normally) on whether it *ever* completes. Common examples include:

|  |  |
| --- | --- |
| **Events** | Mouse clicks, etc. |
| **Notifications** | Status change alerts |
| **Postings** | Mail messages, stock quotes, etc. |
| **Activations** | Creating Applets, daemons, etc. |
| **Commands** | Print requests, etc. |
| **Relays** | Message forwardings and dispatchings |

Oneway interactions between senders and recipients need not be strictly asynchronous. For example, the sender may be responsible for ensuring that a recipient actually receives the message. Also, the sender or another object may later wish to cancel or roll back the effects of the resulting task (which is of course not always possible, for example if the task has already completed — see 3.1.2).

[**Figure 4-2**](javascript:popUp('/content/images/0201310090/elementLinks/04fig02.gif'))

If every task could run instantaneously, you might trigger oneway messages via procedural invocations in which the caller waits out the task triggered by the message, even though it has no reason to do so. But there are often performance-based, conceptual, and logistical reasons to issue some of these messages via thread-based constructions in which the associated tasks proceed independently.

## 4.1.1 Message Formats

Many different styles of invocation are encompassed under the notion of oneway message passing. While some of them are more closely associated with distributed or multiprocess applications (see 1.2.2), any of them can be used in conjunction with the constructions discussed in this section. In addition to direct method invocations, message formats may include:

**Command strings.** The recipient must parse, decode, and then dispatch the associated task. Command string messages are widely used in socket-based and pipe-based communication, especially in web services.

**Event objects.** The message contains a structured description of an event. The recipient then dispatches some arbitrary handling task that it associates with the event. Event objects are used extensively in GUI frameworks such as java.awt, as well as component frameworks supported by java.beans.

**Request objects.** The message contains an encoding of a method name and (*marshalled*or *serialized*) arguments. The recipient issues the corresponding method call to a helper object that performs this method. Request objects are used in distributed object support systems such as those in java.rmi and org.omg.corba. Variants are used in Ada tasking.

**Class objects.** The message is a representation of a class (for example via a .class file) which the recipient then instantiates. This scheme is used in the java.applet framework, as well as in remote activation protocols.

**Runnable objects.** The message consists of some code that the recipient executes. Mixed forms of *runnable events* (which include both an event description and an associated action) are used in some event frameworks. Extended forms employing serialized runnable objects are seen in mobile agent frameworks.

**Arbitrary objects.** A sender may treat any kind of object as a message by including it as method argument or passing it through a Channel (see 4.2.1). For example, in the JavaSpaces™ framework, senders may post any serialized object as a message (also known as an *entry*). Recipients accept only those entries with types and field values that conform to a specified set of matching criteria. Recipients then process these objects in any appropriate manner.

Differences among these formats reflect (among other things) how much the caller knows about the code the recipient needs to run to perform its task. It is often both most convenient and most efficient to use runnable objects, especially in thread-based frameworks that use instances of class Runnable as arguments in Thread constructors. We'll focus on this form, but occasionally illustrate others.

## 4.1.2 Open Calls

Consider the central Host object in a call chain in which the Host receives req requests from any number of Clients and, in the course of processing them, must issue logically oneway handlemessages to one or more Helper objects. Again, we'll ignore the facts that an arbitrary amount of effort might be needed to decode the request before acting upon it, that the request might actually be read from a socket as seen in the WebService class, and so on. Also, all classes discussed in this section can be extended to issue multicasts to multiple helpers using the constructions described in 2.4.4 and 3.5.2.

[**Figure 4-3**](javascript:popUp('/content/images/0201310090/elementLinks/04fig03.gif'))

The main design force here is latency. If a Host is busy servicing requests, then it cannot accept new ones. This adds response time to new requests from Clients, reducing overall service availability.

Some aspects of latency can be addressed simply by using the pass-through and open call designs described in 2.4:

class OpenCallHost { // Generic code sketch

protected long localState;

protected final Helper helper = new Helper();

protected synchronized void updateState(...) {

localState = ...;

}

public void req(...) {

updateState(...);

helper.handle(...);

}

}

Here, even if the helper.handle call is relatively time-consuming, the Host object will still be able to accept new requests from clients running in different threads. The request acceptance rate is bounded only by the time it takes to update local state.

The use of open calls typically eliminates bottleneck points surrounding a given Host, but does not address the broader question of how to introduce concurrency into a system to begin with. Open calls are useful only when clients somehow already know enough to use some other approach that permits independent execution when necessary or desired.

## 4.1.3 Thread-Per-Message

Concurrency can be introduced into oneway messaging designs by issuing a message in its own thread, as in:

class ThreadPerMessageHost { // Generic code sketch

protected long localState;

protected final Helper helper = new Helper();

protected synchronized void updateState() {

localState = ...;

}

public void req(...) {

updateState(...);

new Thread(new Runnable() {

public void run() {

helper.handle(...);

}

}).start();

}

}

[**Figure 4-4**](javascript:popUp('/content/images/0201310090/elementLinks/04fig04.gif'))

This strategy improves throughput when multiple parallel tasks can run faster than a sequence of them could, normally because they are either IO-bound or are compute-bound and running on a multiprocessor. It can also enhance fairness and improve availability if clients need not wait for each other's tasks to complete.

Decisions about whether to create and start threads to perform tasks are not too different from decisions about whether to create other kinds of objects or send other kinds of messages: The benefits must outweigh the costs.

Thread-per-message designs introduce response latency because thread creation is more expensive than direct method invocation. When tasks are time-consuming compared to thread construction time, are session-based, need to be isolated from other independent activities, or can exploit IO or CPU parallelism, the trade-offs are generally worth it. But performance problems can emerge even when construction latencies are acceptable. The JVM implementation and/or operating system may not respond well to the construction of too many threads. For example, they may run out of system resources associated with threads. Also, as the number of threads increases, thread scheduling and context switching overhead can overwhelm processing times.

### 4.1.3.1 Executors

The coding style seen in class ThreadPerMessage can become a problem because of its direct reliance on class Thread. Such usages can make it more difficult to adjust thread initialization parameters, as well as thread-specific data (see 2.3.2) used across an application. This can be avoided by creating an interface, say:

interface Executor {

void execute(Runnable r);

}

This interface can be implemented with classes such as:

class PlainThreadExecutor implements Executor {

public void execute(Runnable r) {

new Thread(r).start();

}

}

These implementations may be used in classes such as:

class HostWithExecutor { // Generic code sketch

protected long localState;

protected final Helper helper = new Helper();

protected final Executor executor;

public HostWithExecutor(Executor e) { executor = e; }

protected synchronized void updateState(...) {

localState = ...;

}

public void req(...) {

updateState(...);

executor.execute(new Runnable() {

public void run() {

helper.handle(...);

}

});

}

}

The use of such interfaces also permits replacement of threads with lightweight executable frameworks.

## 4.1.4 Worker Threads

Lightweight executable frameworks fill the gap between open calls and thread-per-message designs. They apply when you need to introduce limited concurrency, at the expense of some usage restrictions, in order to maximize (or at least improve) throughput and minimize average latencies.

Lightweight executable frameworks can be constructed in many ways, but all stem from the basic idea of using one thread to execute many unrelated tasks (here, in succession). These threads are known as*worker* threads, *background* threads, and as *thread pools* when more than one thread is used.

Each worker continually accepts new Runnable commands from hosts and holds them in some kind ofChannel (a queue, buffer, etc. — see 3.4.1) until they can be run. This design has the classic form of a producer-consumer relationship: the host produces tasks and workers consume them by running them.

[**Figure 4-5**](javascript:popUp('/content/images/0201310090/elementLinks/04fig05.gif'))

Lightweight executable frameworks can improve the structure of some task-based concurrent programs, by allowing you to package many smaller, logically asynchronous units of execution as tasks without having to worry much about performance consequences: Entering a Runnable into a queue is likely to be faster than creating a new Thread object. And because you can control the number of worker threads, you can minimize chances of resource exhaustion and reduce context-switching overhead. Explicit queuing also permits greater flexibility in tuning execution semantics. For example, you can implement Channels as priority queues that order tasks with more deterministic control than is guaranteed by Thread.setPriority. (See 4.3.4 for an example.)

[**Figure 4-6**](javascript:popUp('/content/images/0201310090/elementLinks/04fig06.gif'))

To interoperate with pure thread-based versions, worker threads can be packaged as Executors. Here is a generic implementation that could be used in the HostWithExecutor class instead of the thread-per-message version:

class PlainWorkerPool implements Executor {

protected final Channel workQueue;

public void execute(Runnable r) {

try {

workQueue.put(r);

}

catch (InterruptedException ie) { // postpone response

Thread.currentThread().interrupt();

}

}

public PlainWorkerPool(Channel ch, int nworkers) {

workQueue = ch;

for (int i = 0; i < nworkers; ++i) activate();

}

protected void activate() {

Runnable runLoop = new Runnable() {

public void run() {

try {

for (;;) {

Runnable r = (Runnable)(workQueue.take());

r.run();

}

}

catch (InterruptedException ie) {} // die

}

};

new Thread(runLoop).start();

}

}

### 4.1.4.1 Design choices

The first decision to make surrounding lightweight executable frameworks based on worker threads is whether to create or use them at all. The main question is whether there is some property of ordinaryThreads that you do not need or are willing to give up. If not, it is unlikely that you will arrive at a solution that outperforms the built-in thread support on production JVM implementations.

The trade-offs that obtain the performance advantages of worker threads have several additional tunable parameters, usage consequences, and programming obligations that can impact the design and use of worker thread classes (including those contained in the util.concurrent package available from the online supplement).

#### Identity

Most worker threads must be treated "anonymously". Because the same worker thread is reused for multiple tasks, the use of ThreadLocal and other thread-specific contextual control techniques (see 2.3.2) becomes more awkward. To cope with this, you need to know about all such contextual data, and somehow reset it if necessary upon executing each task. (This includes information about security contexts maintained by run-time support classes.) However, most lightweight executable frameworks avoid any reliance on thread-specific techniques.

If identity is the only property of threads you are willing to give up, then the only potential performance value of worker threads is minimization of start-up overhead by reusing existing threads to execute multiple Runnable tasks, while still possibly bounding resource consumption.

#### Queuing

Runnable tasks that are sitting in queues do not run. This is one source of performance benefits in most worker-thread designs — if each action were associated with a thread, it would need to be independently scheduled by the JVM. But as a consequence, queued execution cannot in general be used when there are any dependencies among tasks. If a currently running task blocks waiting for a condition produced by a task still waiting in the queue, the system may freeze up. Options here include:

* Use as many worker threads as there are simultaneously executing tasks. In this case, theChannel need not perform any queuing, so you can use SynchronousChannels (see 3.4.1.4), queueless channels that require each put to wait for a take and vice versa. Here, the host objects merely hand off tasks to worker threads, which immediately start executing them. For this to work well, worker thread pools should be dynamically expandable.
* Restrict usage to contexts in which task dependencies are impossible, for example in HTTP servers where each message is issued by an unrelated external client requesting a file. Require the helper objects to create actual Threads when they cannot ensure independence.
* Create custom queues that understand the dependencies among the particular kinds of tasks being processed by the worker threads. For example, most pools used for processing tasks representing transactions (see 3.6) must keep track of transaction dependencies. And the lightweight parallel framework described in 4.4.1 relies on special queuing policies that apply only to subtasks created in divide-and-conquer algorithms.

#### Saturation

As the request rate increases, a worker pool will eventually become saturated. All worker threads will be processing tasks and the Host object(s) using the pool will be unable to hand off work. Possible responses include:

* Increase the pool size. In many applications, bounds are heuristic estimates. If a bound is just a guess based on values shown to work well on a particular platform under test workloads, it can be increased. At some point, though, one of the other options must be taken unless you can tolerate failure if the JVM runs out of enough resources to construct a new Thread.
* If the nature of the service allows it, use an unbounded buffered channel and let requests pile up. This risks potential system failure due to exhaustion of memory, but this takes longer to happen than does resource exhaustion surrounding Thread construction.
* Establish a *back-pressure* notification scheme to ask clients to stop sending so many requests. If the ultimate clients are part of a distributed system, they may be able to use another server instead.
* *Drop* (discard) new requests upon saturation. This can be a good option if you know that clients will retry anyway. However, unless retries are automatic, you need to add callbacks, events, or notifications back to clients to alert them of the drops so that they will know enough to retry (see 4.3.1).
* Make room for the new request by dropping *old* requests that have been queued but not yet run, or even cancelling one or more executing tasks. This preference for new requests over old ones upon saturation sometimes meshes well with usage patterns. For example, in some telecommunications systems, old unserviced tasks are usually requests by clients that have already given up and disconnected.
* Block until some thread is available. This can be a good option when handlers are of predictable, short-lived duration, so you can be confident that the wait will unblock without unacceptable delays.
* The Host can run the task directly itself, in its current thread. This is often the best default choice. In essence, the Host momentarily becomes single-threaded. The act of servicing the request limits the rate at which it can accept new requests, thus preventing further local breakdowns.

#### Thread management

The PlainWorkerPool class is somewhat wasteful because it creates all worker threads upon start-up, whether they are needed or not, and lets them all live on indefinitely, even when the service is not being used. These problems can be alleviated by using a management class that supports:

* **Lazy construction:** Activate a new thread only when a request cannot be serviced immediately by an existing idle thread. Lazy construction allows users to provide large enough pool size limits to avoid underutilization problems occurring when fewer threads are running than a given computer can handle. This comes at the minor expense of occasionally higher latencies when a new request causes a new thread to be created. The start-up effects of lazy construction can be tempered by creating a small number of "warm" threads upon construction of the pool.
* **Idle time-outs:** Allow threads to time out waiting for work and to terminate upon time-out. This eventually causes all workers to exit if the pool is not used for prolonged periods. When coupled with lazy construction, these dead threads will be replaced with new ones if the request rate later increases.

In heavily resource-conscious applications, you may also associate other resources (such as sets of reusable graphical objects) with each worker thread, thus combining resource pools (see 3.4.1.2) with thread pools.

#### Cancellation

You may need to distinguish cancellation (see 3.1.2) of a task from cancellation of the worker thread performing that task. One approach is:

* Upon interruption, allow the current worker thread to die, but replace it if necessary with a fresh worker thread if the work queue is not empty or when a new incoming task arrives.
* Provide a shutdown method in the worker thread class that causes existing workers to die and no additional workers to be created.

Additionally, you may need to trigger some kind of error handling if a Host thread is cancelled during a task hand-off. While the silent swallowing of InterruptedException without queuing a task seen inPlainWorkerPool conforms to the minimal requirements of oneway message-passing frameworks, most applications need to take other remedial actions.

### 4.1.4.2 Event queues

Many event-based frameworks (including the ones supported in the java.awt and javax.swingpackages) rely on designs in which exactly one worker thread operates on an unbounded queue. The queue holds instances of EventObject that must be dispatched (as opposed to Runnable objects that self-dispatch), normally to *listener* objects defined by the application. Often the listeners are the same objects as those that initially generate events.

[**Figure 4-7**](javascript:popUp('/content/images/0201310090/elementLinks/04fig07.gif'))

The use of a single thread operating on a single event queue simplifies usage compared to general worker-thread designs, but also imposes some limitations that are characteristic of event frameworks:

* The ordering properties of a queue can be exploited to optimize handling. For example, automatic event-filtering techniques can be used to remove or combine duplicate repaint events for the same screen area before they hit the front of the queue and are taken by the worker thread.
* You can require that all methods operating on certain objects be invoked only by issuing events onto the queue, and are thus ultimately performed by the single worker thread. This results in a form of thread confinement (see 2.3.2) of these objects. If flawlessly adhered to, this eliminates the need for dynamic locking within operations on these objects, thus improving performance. This can also reduce complexity for applications that do not otherwise need to construct threads.

This is the basis for the Swing *single-thread rule*: With only a few exceptions, all manipulation of Swing objects must be performed by the event handler thread. While not stated in AWT, it is good idea to observe this rule there as well.

* Events should not be enabled until their handlers are fully constructed and are thus ready to handle events. This holds as well for other thread-based designs (see 2.2.7), but is a more common source of error here because registering an event handler or listener *inside* its constructor is not as obvious a way to prematurely enable concurrent execution as is constructing a thread.
* Users of the event framework must never dispatch actions that block in ways that can unblock only as a result of handling a future event. This problem is encountered when implementing modal dialogs in most event frameworks, and requires an ad-hoc solution. However, more localized solutions can be obtained merely by setting a *disabled* state for interactive components that should not be used until a certain re-enabling event is received. This avoids blocking the event queue without allowing undesired actions to be triggered.
* Further, to maintain responsiveness of the event framework, actions should not block at all, and should not perform time-consuming operations.

This set of design choices causes event frameworks to have much better performance than would thread-per-event designs, and makes them simpler to program by developers who do not otherwise use threads. However, the usage restrictions have more impact in programs that do construct other threads. For example, because of the single-thread rule, even the smallest manipulations of GUI components (such as changing the text in a label) must be performed by issuing runnable event objects that encapsulate an action to be performed by the event handler thread.

In Swing and AWT applications, the methods javax.swing.SwingUtilities.invokeLater andjava.awt.EventQueue.invokeLater can be used to execute display-related commands in the event handler thread. These methods create runnable event objects that are executed when taken from the queue. The online supplement contains links to a SwingWorker utility class that partially automates conformance to these rules for threads that produce results leading to screen updates.

### 4.1.4.3 Timers

The fact that Runnable tasks in worker thread designs may sit queued without running is a problem to be worked around in some applications. But it sometimes becomes a feature when actions are intended to be delayed.

The use of worker threads can both improve efficiency and simplify usage of delayed and periodic actions — those triggered at certain times, after certain delays, or at regular intervals (for example, every day at noon). A standardized timer facility can both automate messy timing calculations and avoid excess thread construction by reusing worker threads. The main trade-off is that if a worker blocks or takes a long time processing one task, the triggering of others may become delayed longer than they would be if separate Threads are created and scheduled by the underlying JVM.

Time-based daemons can be constructed as variants of the basic worker thread design described in 4.1.4.1. For example, here are the highlights of a version that relies on an unshown priority queue class (that might take a form similar to the scheduling queue illustrated in 4.3.4) and is set up to support only one worker thread:

class TimerDaemon { // Fragments

static class TimerTask implements Comparable { // ...

final Runnable command;

final long execTime; // time to run at

public int compareTo(Object x) {

long otherExecTime = ((TimerTask)(x)).execTime;

return (execTime < otherExecTime) ? -1 :

(execTime == otherExecTime)? 0 : 1;

}

}

// a heap or list with methods that preserve

// ordering with respect to TimerTask.compareTo

static class PriorityQueue {

void put(TimerTask t);

TimerTask least();

void removeLeast();

boolean isEmpty();

}

protected final PriorityQueue pq = new PriorityQueue();

public synchronized void executeAfterDelay(Runnable r,long t){

pq.put(new TimerTask(r, t + System.currentTimeMillis()));

notifyAll();

}

public synchronized void executeAt(Runnable r, Date time) {

pq.put(new TimerTask(r, time.getTime()));

notifyAll();

}

// wait for and then return next task to run

protected synchronized Runnable take()

throws InterruptedException {

for (;;) {

while (pq.isEmpty())

wait();

TimerTask t = pq.least();

long now = System.currentTimeMillis();

long waitTime = now - t.execTime;

if (waitTime <= 0) {

pq.removeLeast();

return t.command;

}

else

wait(waitTime);

}

}

public TimerDaemon() { activate(); } // only one

void activate() {

// same as PlainWorkerThread except using above take method

}

}

The techniques discussed in 3.7 can be used here to improve efficiency of the waiting and notification operations.

This class can be extended to deal with periodic tasks by including additional bookkeeping to requeue them before running them. However, this also requires dealing with the fact that periodically scheduled actions are almost never exactly periodic, in part because timed waits do not necessarily wake up exactly upon the given delays. The main options are either to ignore lags and reschedule by clock time, or to ignore the clock and reschedule the next execution at a fixed delay after starting the current one. Fancier schemes are typically needed for multimedia synchronization — see the Further Readings in 1.3.5.

Timer daemons1 can additionally support methods that cancel delayed or periodic actions. One approach is to have executeAt and other scheduling methods accept or return suitably a reworkedTimerTask supporting a cancel method that sets a status flag honored by the worker thread.

## 4.1.5 Polling and Event-Driven IO

Most worker thread designs rely on blocking channels in which the worker thread waits for incoming commands to run. However, there are a few contexts in which optimistic-style retry loops provide a better solution. Most involve the execution of commands stemming from messages received across IO streams.

It can be a challenge to achieve low latencies and high throughputs in heavily loaded IO-bound systems. The time taken to create a thread that performs an IO-based task adds latency, but most run-time systems are tuned such that, once threads are created, they are very responsive to new inputs arriving on IO streams. On input, they unblock with shorter latencies than you are likely to achieve via other techniques. Especially in the case of socket-based IO, these forces generally favor thread-per-IO-session designs, where a different thread is used (or reused) for each session relying on input from a different connection.

[**Figure 4-8**](javascript:popUp('/content/images/0201310090/elementLinks/04fig08.gif'))

However, as the number of simultaneously active connections climbs, other approaches are (only) sometimes more attractive. Consider for example, a multiplayer game server, or a transaction server, with:

* Thousands of simultaneous socket connections that join and leave at a steady rate, for example, as people start and finish playing a game.
* Relatively low input rates on any given socket at any given time. However, summing across all connections, the aggregate IO rates may be very high.
* Non-trivial computation associated with at least some inputs, for example those that cause global state changes in games.

On large mainframe systems, this kind of problem is sometimes dealt with by creating a special-purpose front-end machine that multiplexes all of the inputs into a single stream that is then dealt with by the main service. The main service is often multithreaded, but its structure is simplified and made more efficient because it does not need to deal with so many apparent clients at a time.

A family of polling and event-driven designs approach such problems without requiring special front ends. While they are not (as of this writing) explicitly supported by the java.io and java.net classes, enough of the ingredients are provided to allow construction of designs that can attain good performance in these kinds of situations. (The designs are analogous to those using socket select andpoll operations in other systems and languages.) We'll illustrate with inputs on sockets, but the approach also applies to outputs, to files, and to IO using more exotic devices such as sensors.

### 4.1.5.1 Event-driven tasks

Many IO-based tasks are initially written in a session-based style (see 2.3.1), continuously pulling commands from sockets and processing them. For example:

class SessionTask implements Runnable { // Generic code sketch

protected final Socket socket;

protected final InputStream input;

SessionTask(Socket s) throws IOException {

socket = s; input = socket.getInputStream();

}

public void run() { // Normally run in a new thread

byte[ ] commandBuffer = new byte[BUFFSIZE];

try {

for (;;) {

int bytes = input.read(commandBuffer, 0, BUFFSIZE);

if (bytes != BUFFSIZE) break;

processCommand(commandBuffer, bytes);

}

}

catch (IOException ex) {

cleanup();

}

finally {

try { input.close(); socket.close(); }

catch(IOException ignore) {}

}

}

}

To enable many sessions to be handled without using many threads, the tasks first must be refactored into an event-driven style, where an event here signifies IO availability. In this style, a session consists of possibly many executions of its event-triggered task(s), each of which is invoked when input becomes available. Event-driven IO tasks are similar in form to GUI event handlers. A session-based design can be converted into an event-driven form by:

* Isolating the basic per-command functionality in a reworked task run method that reads *one*command and performs the associated action.
* Defining the run method so that it can be *repeatedly* triggered whenever input is available to be read (or an IO exception occurs).
* Manually maintaining completion status so that the per-event action is no longer triggered when the session finishes, normally because the input has been exhausted or the connection has been closed.

For example:

class IOEventTask implements Runnable { // Generic code sketch

protected final Socket socket;

protected final InputStream input;

protected volatile boolean done = false; // latches true

IOEventTask(Socket s) throws IOException {

socket = s; input = socket.getInputStream();

}

public void run() { // trigger only when input available

if (done) return;

byte[ ] commandBuffer = new byte[BUFFSIZE];

try {

int bytes = input.read(commandBuffer, 0, BUFFSIZE);

if (bytes != BUFFSIZE) done = true;

else processCommand(commandBuffer, bytes);

}

catch (IOException ex) {

cleanup();

done = true;

}

finally {

if (!done) return;

try { input.close(); socket.close(); }

catch(IOException ignore) {}

}

}

// Accessor methods needed by triggering agent:

boolean done() { return done; }

InputStream input() { return input; }

}

#### 4.1.5.2 Triggering

When the events driving each event-driven task are relatively infrequent, a large number of tasks can be processed by a small number of worker threads. The simplest case occurs when the number of worker threads is exactly one. Here, the worker thread repeatedly polls a list of open sockets to see if they have any input available (via InputStream.available) or have encountered other IO-related status changes. If so, the worker executes the associated run method.

This style of worker thread differs from the ones in 4.1.4.1 in that, rather than pulling tasks from a blocking queue and blindly running them, the worker must repeatedly check a list of registered tasks to see if any can be run. It removes each task from the list only when it claims to have completed.

One generic form is:

class PollingWorker implements Runnable { // Incomplete

private List tasks = ...;

private long sleepTime = ...;

void register(IOEventTask t) { tasks.add(t); }

void deregister(IOEventTask t) { tasks.remove(t); }

public void run() {

try {

for (;;) {

for (Iterator it = tasks.iterator(); it.hasNext();) {

IOEventTask t = (IOEventTask)(it.next());

if (t.done())

deregister(t);

else {

boolean trigger;

try {

trigger = t.input().available() > 0;

}

catch (IOException ex) {

trigger = true; // trigger if exception on check

}

if (trigger)

t.run();

}

}

Thread.sleep(sleepTime); // pause between sweeps

}

}

catch (InterruptedException ie) {}

}

}

Several design concerns arise here:

* Polling intrinsically relies on busy-wait loops (see 3.2.6), which are intrinsically wasteful (but still sometimes less so than context-switching). Coping with this requires empirically guided decisions about how to insert sleeps, yields, or alternative actions to strike a balance between conserving CPU time and maintaining acceptable average response latencies.
* Performance is very sensitive to the characteristics of the underlying data structure maintaining the list of registered tasks. If new tasks come and go regularly, the list of tasks can change fairly frequently. In this case, schemes such as copy-on-write (see 2.4.4) usually do not work well. But there is every reason to make traversal of the list as cheap as possible. One approach is to maintain a cached list for traversal and to update it (if necessary) only at the end of each sweep.
* Event-driven tasks should be triggered only when they have enough data to perform their associated actions. However, in many applications (for example those using free-form string-based commands), the minimal amount of data needed for triggering is not known in advance. In practice (as illustrated here), it usually suffices just to check that at least one byte is available. This exploits the fact that socket-based clients send packets — normally each packet contains an entire command. However, when commands do not arrive as units, the worker thread can stall, thus increasing latencies of other tasks unless buffering schemes are added.
* A single worker thread is not likely to be acceptable if some inputs lead to time-consuming computations or blocking IO. One solution is to require that such computations be performed in new threads or by separate worker thread pools. However, it is sometimes more efficient instead to employ multiple polling worker threads; enough so that on average there will always be a thread polling for inputs.
* The use of multiple polling worker threads requires additional coordination to make sure that two workers are not both trying to run the same task at the same time, without otherwise impeding each other's sweeps through the list of tasks. One approach is to have task classes set and honor *busy* status, for example, via testAndSet (see 3.5.1.4).

Given these concerns and the context dependence of the associated design decisions, it is not surprising that most frameworks are custom-built to suit the demands of particular applications. However, the util.concurrent package available from the online supplement includes some utilities that can be used to help build standardized solutions.

### 4.1.6 Further Readings

Most details about messages, formats, transports, etc., used in practice are specific to particular packages and systems, so the best sources are their accompanying manuals and documentation.

Discussions of message passing in distributed systems can be found in the sources listed in 1.2.5. Any of several packages and frameworks can be used to extend the techniques discussed here to apply in distributed contexts. For example, most of these designs (as well as most in 4.2 and elsewhere in this book) can be adapted for use in JavaSpaces. Conversely, many distributed message passing techniques can be scaled down to apply in concurrent, non-distributed settings.

Design and implementation using JavaSpaces is discussed in:

Freeman, Eric, Susan Hupfer, and Ken Arnold. *JavaSpaces™: Principles, Patterns, and Practice*, Addison-Wesley, 1999.

For different approaches, see for example the Aleph, JMS, and Ninja packages, accessible via links from the online supplement. Many commercial distributed systems are based on CORBA and related frameworks, which also include some support for oneway message passing. See:

Henning, Michi, and Steve Vinoski. *Advanced CORBA Programming with C++*, Addison-Wesley, 1999.

Pope, Alan. *The CORBA Reference Guide*, Addison-Wesley, 1998.

Some systems-level oneway messaging strategies otherwise similar to those presented here are described in:

Langendoen, Koen, Raoul Bhoedjang, and Henri Bal. "Models for Asynchronous Message Handling", *IEEE Concurrency*, April-June 1997.

An argument that single-queue, single-thread event frameworks are a better basis for application programming than thread-based frameworks may be found in:

Ousterhout, John. "Why Threads Are a Bad Idea (For Most Purposes)", *USENIX Technical Conference*, 1996.

[Home](http://www.informit.com/) > [Articles](http://www.informit.com/articles/index.aspx) > [Programming](http://www.informit.com/articles/index.aspx?st=60206) > [Java](http://www.informit.com/articles/index.aspx?st=60209)

* By [Doug Lea](http://www.informit.com/authors/bio/96c55a18-b863-44c2-8a0c-d64bc6822cd2)
* Jun 1, 2001

[📄 Contents](http://www.informit.com/articles/article.aspx?p=167821&seqNum=3)

[␡](http://www.informit.com/articles/article.aspx?p=167821&seqNum=3)

1. [1 Oneway Messages](http://www.informit.com/articles/article.aspx?p=167821)
2. 2 Composing Oneway Messages
3. [3 Services in Threads](http://www.informit.com/articles/article.aspx?p=167821&seqNum=4)
4. [4 Parallel Decomposition](http://www.informit.com/articles/article.aspx?p=167821&seqNum=5)
5. [5 Active Objects](http://www.informit.com/articles/article.aspx?p=167821&seqNum=6)

* [⎙ Print](http://www.informit.com/articles/printerfriendly/167821)
* [+ Share This](http://www.addthis.com/bookmark.php)
* [💬 Discuss](http://www.informit.com/articles/article.aspx?p=167821&seqNum=3#articleDiscussion)

[< Back](http://www.informit.com/articles/article.aspx?p=167821) **Page 2** of 5 [Next >](http://www.informit.com/articles/article.aspx?p=167821&seqNum=4)

## 4.2 Composing Oneway Messages

Many interprocess and distributed designs involve groups of objects exchanging oneway messages (see 1.2 and 4.5). Similar techniques may be applied within individual concurrent programs. In fact, as discussed in 4.1, a larger range of design options is available in concurrent programs than in distributed systems. Messages need not be restricted to, say, socket-based commands. Concurrent programs may also employ lighter alternatives including direct invocations and event-based communication.

However, this wide range of options also introduces opportunities for creating chaotic, difficult-to-understand designs. This section describes some simple program-level (or subsystem-level) structuring techniques that tend to produce well-behaved, readily understandable, and readily extensible designs.

A *flow network* is a collection of objects that all pass oneway messages transferring information and/or objects to each other along paths from *sources* to *sinks*. Flow patterns may occur in any kind of system or subsystem supporting one or more series of connected steps or *stages*, in which each stage plays the role of a *producer* and/or *consumer*. Broad categories include:

**Control systems**. External sensor inputs ultimately cause control systems to generate particular effector outputs. Applications such as avionics control systems contain dozens of kinds of inputs and outputs. For a plainer example, consider a skeletal thermostatic heater control:

[**Figure 4-9**](javascript:popUp('/content/images/0201310090/elementLinks/04fig09.gif'))

**Assembly systems.** Newly created objects undergo a series of changes and/or become integrated with other new objects before finally being used for some purpose; for example, an assembly line for Cartons:

[**Figure 4-10**](javascript:popUp('/content/images/0201310090/elementLinks/04fig10.gif'))

**Dataflow systems.** Each stage transforms or otherwise processes data. For example, in pipelined multimedia systems, audio and/or video data is processed across multiple stages. In *publish-subscribe* systems, possibly many data sources send information to possibly many consumers. In Unix pipes-and-filters shell programs, stages send character data, as in a simple spell checker:

[**Figure 4-11**](javascript:popUp('/content/images/0201310090/elementLinks/04fig11.gif'))

**Workflow systems.** Each stage represents an action that needs to be performed according to some set of business policies or other requirements; for example, a simple payment system:

[**Figure 4-12**](javascript:popUp('/content/images/0201310090/elementLinks/04fig12.gif'))

**Event systems.** Stages pass around and ultimately execute code associated with objects representing messages, user inputs, or simulated physical phenomena. The beginnings of many event systems take the form:

[**Figure 4-13**](javascript:popUp('/content/images/0201310090/elementLinks/04fig13.gif'))

## 4.2.1 Composition

The development of flow networks entails two main sets of concerns: design of the data being passed around, and design of the stages that do the passing.

### 4.2.1.1 Representations

Flow networks pass around representational components — families of values or objects representing the things the flow is all about. In the introductory examples, temperatures, cardboard sheets, words, invoices, and events are the basic kinds of values and objects passed across connected stages. Often these components are interesting objects in their own rights that can perform services, communicate with other objects, and so on. But when viewed as the raw material for a flow, they are treated as mere passive representations, providing data or information rather than behavior.

While they play similar roles in the overall design of a flow system, different categories of representation types affect the details of the rest of the system:

* Information types representing the state of the world (for example values such as temperature readings, maintained as scalars or immutable ADT objects) differ from most others in that it is often acceptable to reuse old or current best-estimate values if necessary. In essence, producers have an inexhaustible supply of such values.
* Event indicators normally can be used at most once, although they may be passed around many times before being used.
* Mutable resource types (such as cartons) may be *transferred* (see 2.3.4) from each stage to the next, ensuring that each object is being operated upon by at most one stage at any given time.
* Alternatively, if the identities of mutable representation objects do not matter, they can be copied across stages as needed. Copy-based approaches are more often used in distributed flow networks in which ownership cannot be transferred across stages simply by assigning reference fields.
* Artificial data types can be used for control purposes. For example, a special *null* token may be used as a terminator that triggers cancellation and shutdown. Similarly, a special*keepalive* can be sent to inform one stage that another still exists. Alternatively, a distinct set of *sideband* control methods can be employed across stages. Sideband controls are methods used to set stages into different modes that influence their main processing. For example, a thermostat Comparator may have a separate control to change its threshold.

### 4.2.1.2 Stages

Stages in well-behaved flow networks all obey sets of constraints that are reminiscent of those seen in electrical circuit design. Here is one conservative set of composition rules that generate a small number of basic kinds of stages:

**Directionality.** Flow maintains a single directionality, from sources to sinks. There are no loops or back-branches from consumers to producers. This results in a directed acyclic graph (DAG) of information or object flow.

**Interoperability.** Methods and message formats are standardized across components, normally through conformance to a small set of interfaces.

**Connectivity.** Stages maintain fixed connectivity: consumers may receive messages only from known producers, and vice versa. So, for example, while a web service may have any number of anonymous clients, a given TemperatureComparator object may be designed to receive temperature update messages only from a designated TemperatureSensor object.

Connectivity is usually arranged by maintaining direct references from producers to consumers or vice versa, or by having them share access to a Channel. Alternatively, a network may be based on constrained use of blackboards, multicast channels, or JavaSpaces (see 4.1.6) in which producers specially tag messages destined for particular consumers.

**Transfer protocols.** Every message transfers information or objects. Once a stage has transferred a mutable object, it never again manipulates that object. When necessary, special buffer stages may be interposed to hold elements transferred out from one stage that cannot yet be accepted by other stages.

Transfer protocols typically rely on the basic put and take operations described in 2.3.4. When all messages involve put-based transfers, networks are normally labeled as *push*flow; when they involve take-based transfers, they are normally labeled as *pull* flow; when they involve channels supporting both put and take (and possibly exchange), they can take various mixed forms.

**Threads.** Stages may implement oneway message passing using any of the patterns described in 4.1, as long as every (potentially) simultaneously live connection from a given producer to a given consumer employs a different thread or thread-based message-sending construction.

It is rarely necessary to satisfy this requirement by issuing every message, or every stream of messages from a producer to a consumer, in a different thread. You can instead exploit connectivity rules to use threads only as needed. Most sources in push-based systems intrinsically employ threads. Additionally, any push stage with multiple successors that may ultimately hit a Combiner stage must issue the messages independently. Otherwise, if a thread is blocked at the combine point, there may be a possibility that the Combiner will never see the other inputs necessary to unblock it.

Conversely, most sinks in pull-based systems intrinsically employ thread-based message constructions, as do stages involved in split/join connections proceeding from the opposite direction pictured above.

|  |  |
| --- | --- |
| *Sources* have no predecessors. | [Icon1](http://www.informit.com/content/images/0201310090/elementLinks/04icon01.gif) |
| *Sinks* have no successors. | [Icon2](http://www.informit.com/content/images/0201310090/elementLinks/04icon02.gif) |
| *Linear* stages have at most one predecessor and one successor. | [Icon3](http://www.informit.com/content/images/0201310090/elementLinks/04icon03.gif) |
| *Routers* send a message to one of their successors. | [Icon4](http://www.informit.com/content/images/0201310090/elementLinks/04icon04.gif) |
| *Multicasters* send messages to all their successors. | [Icon5](http://www.informit.com/content/images/0201310090/elementLinks/04icon05.gif) |
| *Collectors* accept messages from one of their predecessors at a time. | [Icon6](http://www.informit.com/content/images/0201310090/elementLinks/04icon06.gif) |
| *Combiners* require messages from all their predecessors. | [Icon7](http://www.informit.com/content/images/0201310090/elementLinks/04icon07.gif) |

[**Figure 4-14**](javascript:popUp('/content/images/0201310090/elementLinks/04fig14.gif'))

These rules can be liberalized in various ways. In fact, you can adopt any set of composition rules you like. But the listed constraints serve to eliminate large classes of safety and liveness problems while also satisfying common reusability and performance goals: unidirectional flow avoids deadlock, connectivity management avoids unwanted interleavings across different flows, transfer protocols avoid safety problems due to inadvertent sharing without the need for extensive dynamic synchronization, and interface conformance assures type safety while still permitting interoperability among components.

### 4.2.1.3 Scripting

Adoption of standard set of composition rules makes possible the construction of higher-level tools that arrange for stages to operate *cooperatively*, without otherwise imposing centralized dynamic synchronization control. Composition of flow networks can be treated as a form of *scripting* in the usual sense of the word — semi-automated programming of the code that glues together instances of existing object types. This is the kind of programming associated with languages such as JavaScript, Visual Basic, Unix shells, and FlowMark (a workflow tool). Development of a scripting tool, or integration with an existing one, is an optional step in building systems based around flows.

This architecture is analogous to that of GUI builders consisting of a base set of widgets, packers and layout managers, code to instantiate a particular GUI, and a visual scripter that helps set it all up. Alternatively, it may be possible to script flows through direct manipulation tools by which, for example, components communicate instantly once dragged-and-dropped to connect with others.

## 4.2.2 Assembly Line

The remainder of this section illustrates the design and implementation of flow systems via an example assembly line applet that builds series of "paintings" in a style vaguely reminiscent of the artists Piet Mondrian and Mark Rothko. Only the principal classes are given here. Some include unimplemented method declarations. The full code may be found in the online supplement, which also includes other application-level examples of flow-based systems.

[**Figure 4-15**](javascript:popUp('/content/images/0201310090/elementLinks/04fig15.gif'))

### 4.2.2.1 Representations

To start out, we need some base representation types. In this system, all elements can be defined as subclasses of abstract class Box, where every Box has a color and a size, can display itself when asked, and can be made to deeply clone (duplicate) itself. The color mechanics are default-implemented. Others are left abstract, to be defined differently in different subclasses:

abstract class Box {

protected Color color = Color.white;

public synchronized Color getColor() { return color; }

public synchronized void setColor(Color c) { color = c; }

public abstract java.awt.Dimension size();

public abstract Box duplicate(); // clone

public abstract void show(Graphics g, Point origin);// display

}

The overall theme of this example is to start off with sources that produce simple basic boxes, and then push them through stages that paint, join, flip, and embed them to form the paintings. BasicBoxes are the raw material:

class BasicBox extends Box {

protected final Dimension size;

public BasicBox(int xdim, int ydim) {

size = new Dimension(xdim, ydim);

}

public synchronized Dimension size() { return size; }

public void show(Graphics g, Point origin) {

g.setColor(getColor());

g.fillRect(origin.x, origin.y, size.width, size.height);

}

public synchronized Box duplicate() {

Box p = new BasicBox(size.width, size.height);

p.setColor(getColor());

return p;

}

}

Two fancier kinds of boxes can be made by joining two existing boxes side by side and adding a line-based border surrounding them. Joined boxes can also flip themselves. All this can be done either horizontally or vertically. The two resulting classes can be made subclasses of JoinedPair to allow sharing of some common code:

abstract class JoinedPair extends Box {

protected Box fst; // one of the boxes

protected Box snd; // the other one

protected JoinedPair(Box a, Box b) {

fst = a;

snd = b;

}

public synchronized void flip() { // swap fst/snd

Box tmp = fst; fst = snd; snd = tmp;

}

// other internal helper methods

}

class HorizontallyJoinedPair extends JoinedPair {

public HorizontallyJoinedPair(Box l, Box r) {

super(l, r);

}

public synchronized Box duplicate() {

HorizontallyJoinedPair p =

new HorizontallyJoinedPair(fst.duplicate(),

snd.duplicate());

p.setColor(getColor());

return p;

}

// ... other implementations of abstract Box methods

}

class VerticallyJoinedPair extends JoinedPair {

// similar

The final kind of fancy box wraps one Box within a border:

class WrappedBox extends Box {

protected Dimension wrapperSize;

protected Box inner;

public WrappedBox(Box innerBox, Dimension size) {

inner = innerBox;

wrapperSize = size;

}

// ... other implementations of abstract Box methods

}

### 4.2.2.2 Interfaces

Looking ahead to how we might want to string stages together, it is worthwhile to standardize interfaces. We'd like to be able to connect any stage to any other stage for which it could make sense, so we want bland, noncommittal names for the principal methods.

Since we are doing oneway push-based flow, these interfaces mainly describe put-style methods. In fact, we could just call them all put, except that this doesn't work very well for two-input stages. For example, a VerticalJoiner needs two put methods, one supplying the top Box and one the bottomBox. We could avoid this by designing Joiners to take alternate inputs as the tops and bottoms, but this would make them harder to control. Instead, we'll use the somewhat ugly but easily extensible names putA, putB, and so on:

[**Figure 4-16**](javascript:popUp('/content/images/0201310090/elementLinks/04fig16.gif'))

interface PushSource {

void produce();

}

[**Figure 4-17**](javascript:popUp('/content/images/0201310090/elementLinks/04fig17.gif'))

interface PushStage {

void putA(Box p);

}

[**Figure 4-18**](javascript:popUp('/content/images/0201310090/elementLinks/04fig18.gif'))

interface DualInputPushStage extends PushStage {

void putB(Box p);

}

### 4.2.2.3 Adapters

We can make the "B" channels of DualInputPushStages completely transparent to other stages by defining a simple Adapter class that accepts a putA but relays it to the intended recipient's putB. In this way, most stages can be built to invoke putA without knowing or caring that the box is being fed into some successor's B channel:

[**Figure 4-19**](javascript:popUp('/content/images/0201310090/elementLinks/04fig19.gif'))

class DualInputAdapter implements PushStage {

protected final DualInputPushStage stage;

public DualInputAdapter(DualInputPushStage s) { stage = s; }

public void putA(Box p) { stage.putB(p); }

}

### 4.2.2.4 Sinks

Sinks have no successors. The simplest kind of sink doesn't even process its input, and thus serves as a way to throw away elements. In the spirit of Unix pipes and filters, we can call it:

[**Figure 4-20**](javascript:popUp('/content/images/0201310090/elementLinks/04fig20.gif'))

class DevNull implements PushStage {

public void putA(Box p) { }

}

More interesting sinks require more interesting code. For example, in the applet used to produce the image shown at the beginning of this section, the Applet subclass itself was defined to implementPushStage. It served as the ultimate sink by displaying the assembled objects.

### 4.2.2.5 Connections

Interfaces standardize on the method names for stages but do nothing about the linkages to successors, which must be maintained using some kind of instance variables in each stage object. Except for sinks such as DevNull, each stage has at least one successor. There are several implementation options, including:

* Have each object maintain a collection object holding all its successors.
* Use a master connection registry that each stage interacts with to find out its successor(s).
* Create the minimal representation: define a base class for stages with exactly one successor and one for those with exactly two successors.

The third option is simplest and works fine here. (In fact, it is always a valid option. Stages with three or more outputs can be built by cascading those for only two. Of course, you wouldn't want to do this if most stages had large and/or variable numbers of successors.)

This leads to base classes that support either one or two links and have one or two corresponding attachment methods, named using a similar ugly suffix convention (attach1, attach2). Because connections are dynamically assignable, they are accessed only under synchronization:

[**Figure 4-21**](javascript:popUp('/content/images/0201310090/elementLinks/04fig21.gif'))

class SingleOutputPushStage {

private PushStage next1 = null;

protected synchronized PushStage next1() { return next1; }

public synchronized void attach1(PushStage s) { next1 = s; }

}

[**Figure 4-22**](javascript:popUp('/content/images/0201310090/elementLinks/04fig22.gif'))

class DualOutputPushStage extends SingleOutputPushStage {

private PushStage next2 = null;

protected synchronized PushStage next2() { return next2; }

public synchronized void attach2(PushStage s) { next2 = s; }

}

### 4.2.2.6 Linear stages

Now we can build all sorts of classes that extend either of the base classes, simultaneously implementing any of the standard interfaces. The simplest transformational stages are linear, single-input/single-output stages. Painters, Wrappers, and Flippers are merely:

[**Figure 4-23**](javascript:popUp('/content/images/0201310090/elementLinks/04fig23.gif'))

class Painter extends SingleOutputPushStage

implements PushStage {

protected final Color color; // the color to paint boxes

public Painter(Color c) { color = c; }

public void putA(Box p) {

p.setColor(color);

next1().putA(p);

}

}

[**Figure 4-24**](javascript:popUp('/content/images/0201310090/elementLinks/04fig24.gif'))

class Wrapper extends SingleOutputPushStage

implements PushStage {

protected final int thickness;

public Wrapper(int t) { thickness = t; }

public void putA(Box p) {

Dimension d = new Dimension(thickness, thickness);

next1().putA(new WrappedBox(p, d));

}

}

[**Figure 4-25**](javascript:popUp('/content/images/0201310090/elementLinks/04fig25.gif'))

class Flipper extends SingleOutputPushStage

implements PushStage {

public void putA(Box p) {

if (p instanceof JoinedPair)

((JoinedPair) p).flip();

next1().putA(p);

}

}

Painter and Wrapper stages apply to any kind of Box. But Flippers only make sense forJoinedPairs: if a Flipper receives something other than a JoinedPair, it just passes it through. In a more "strongly typed" version, we might instead choose to drop boxes other than JoinedPairs, perhaps by sending them to DevNull.

### 4.2.2.7 Combiners

We have two kinds of Combiners, horizontal and vertical Joiners. Like the representation classes, these classes have enough in common to factor out a superclass. Joiner stages block further inputs until they can combine one item each from putA and putB. This can be implemented via guard mechanics that hold up acceptance of additional items from putA until existing ones have been paired up with those from putB, and vice versa:

[**Figure 4-26**](javascript:popUp('/content/images/0201310090/elementLinks/04fig26.gif'))

abstract class Joiner extends SingleOutputPushStage

implements DualInputPushStage {

protected Box a = null; // incoming from putA

protected Box b = null; // incoming from putB

protected abstract Box join(Box p, Box q);

protected synchronized Box joinFromA(Box p) {

while (a != null) // wait until last consumed

try { wait(); }

catch (InterruptedException e) { return null; }

a = p;

return tryJoin();

}

protected synchronized Box joinFromB(Box p) { // symmetrical

while (b != null)

try { wait(); }

catch (InterruptedException ie) { return null; }

b = p;

return tryJoin();

}

protected synchronized Box tryJoin() {

if (a == null || b == null) return null; // cannot join

Box joined = join(a, b); // make combined box

a = b = null; // forget old boxes

notifyAll(); // allow new puts

return joined;

}

public void putA(Box p) {

Box j = joinFromA(p);

if (j != null) next1().putA(j);

}

public void putB(Box p) {

Box j = joinFromB(p);

if (j != null) next1().putA(j);

}

}

class HorizontalJoiner extends Joiner {

protected Box join(Box p, Box q) {

return new HorizontallyJoinedPair(p, q);

}

}

class VerticalJoiner extends Joiner {

protected Box join(Box p, Box q) {

return new VerticallyJoinedPair(p, q);

}

}

### 4.2.2.8 Collectors

A Collector accepts messages on either channel and relays them to a single successor:

[**Figure 4-27**](javascript:popUp('/content/images/0201310090/elementLinks/04fig27.gif'))

class Collector extends SingleOutputPushStage

implements DualInputPushStage {

public void putA(Box p) { next1().putA(p);}

public void putB(Box p) { next1().putA(p); }

}

If for some reason we needed to impose a bottleneck here, we could define an alternative form of collector in which these methods are declared as synchronized. This could also be used to guarantee that at most one activity is progressing through a given collector at any given time.

### 4.2.2.9 Dual output stages

Our multiple-output stages should generate threads or use one of the other options discussed in 4.1 to drive at least one of their outputs (it doesn't matter which). This maintains liveness when elements are ultimately passed to Combiner stages (here, the Joiners). For simplicity of illustration, the following classes create new Threads. Alternatively, we could set up a simple worker thread pool to process these messages.

Alternators output alternate inputs to alternate successors:

[**Figure 4-28**](javascript:popUp('/content/images/0201310090/elementLinks/04fig28.gif'))

class Alternator extends DualOutputPushStage

implements PushStage {

protected boolean outTo2 = false; // control alternation

protected synchronized boolean testAndInvert() {

boolean b = outTo2;

outTo2 = !outTo2;

return b;

}

public void putA(final Box p) {

if (testAndInvert())

next1().putA(p);

else {

new Thread(new Runnable() {

public void run() { next2().putA(p); }

}).start();

}

}

}

Cloners multicast the same element to both successors:

[**Figure 4-29**](javascript:popUp('/content/images/0201310090/elementLinks/04fig29.gif'))

class Cloner extends DualOutputPushStage

implements PushStage {

public void putA(Box p) {

final Box p2 = p.duplicate();

next1().putA(p);

new Thread(new Runnable() {

public void run() { next2().putA(p2); }

}).start();

}

}

A Screener is a stage that directs all inputs obeying some predicate to one channel, and all others to the other:

[**Figure 4-30**](javascript:popUp('/content/images/0201310090/elementLinks/04fig30.gif'))

We can build a generic Screener by encapsulating the BoxPredicate to check in an interface and implementing it, for example, with a class that makes sure that a Box fits within a given (symmetric, in this case) bound. The Screener itself accepts a BoxPredicate and uses it to direct outputs:

interface BoxPredicate {

boolean test(Box p);

}

class MaxSizePredicate implements BoxPredicate {

protected final int max; // max size to let through

public MaxSizePredicate(int maximum) { max = maximum; }

public boolean test(Box p) {

return p.size().height <= max && p.size().width <= max;

}

}

class Screener extends DualOutputPushStage

implements PushStage {

protected final BoxPredicate predicate;

public Screener(BoxPredicate p) { predicate = p; }

public void putA(final Box p) {

if (predicate.test(p)) {

new Thread(new Runnable() {

public void run() { next1().putA(p); }

}).start();

}

else

next2().putA(p);

}

}

### 4.2.2.10 Sources

Here is a sample source, one that produces BasicBoxes of random sizes. For convenience, it is also equipped with an autonomous loop run method repeatedly invoking produce, interspersed with random production delays:

[**Figure 4-31**](javascript:popUp('/content/images/0201310090/elementLinks/04fig31.gif'))

class BasicBoxSource extends SingleOutputPushStage

implements PushSource, Runnable {

protected final Dimension size; // maximum sizes

protected final int productionTime; // simulated delay

public BasicBoxSource(Dimension s, int delay) {

size = s;

productionTime = delay;

}

protected Box makeBox() {

return new BasicBox((int)(Math.random() \* size.width) + 1,

(int)(Math.random() \* size.height) + 1);

}

public void produce() {

next1().putA(makeBox());

}

public void run() {

try {

for (;;) {

produce();

Thread.sleep((int)(Math.random() \* 2\* productionTime));

}

}

catch (InterruptedException ie) { } // die

}

}

### 4.2.2.11 Coordination

Without a scripting tool based on these classes, we have to program assembly lines by manually creating instances of desired stages and linking them together. This is easy in principle, but tedious and error-prone in practice because of the lack of visual guidance about what stages are connected to what.

Here's a fragment of the flow used in the applet that produced the image displayed at the beginning of this section:

[**Figure 4-32**](javascript:popUp('/content/images/0201310090/elementLinks/04fig32.gif'))

The code setting this up may be found in the online supplement. The main constructor mostly consists of many lines of the form:

Stage aStage = new Stage();

aStage.attach(anotherStage);

This is followed by invoking start on threads running all the sources.

## 4.2.3 Further Readings

Flow patterns often serve as the computational versions of use cases, scenarios, scripts, and related concepts from high-level object-oriented analysis. Most of the books on OO design and on design patterns listed in 1.3.5 and 1.4.5 describe issues relevant to the analysis, design and implementation of flow-based systems. Domain-specific issues surrounding packet networking, telecommunications, and multimedia systems often requiring more elaborate flow-based designs are discussed in the texts on concurrent and distributed systems in 1.2.5.

[< Back](http://www.informit.com/articles/article.aspx?p=167821) **Page 2** of 5 [Next >](http://www.informit.com/articles/article.aspx?p=167821&seqNum=4)

* [+ Share This](http://www.addthis.com/bookmark.php)
* [🔖 Save To Your Account](http://www.informit.com/articles/article.aspx?p=167821&seqNum=3#addToWishList)

[Home](http://www.informit.com/) > [Articles](http://www.informit.com/articles/index.aspx) > [Programming](http://www.informit.com/articles/index.aspx?st=60206) > [Java](http://www.informit.com/articles/index.aspx?st=60209)

* By [Doug Lea](http://www.informit.com/authors/bio/96c55a18-b863-44c2-8a0c-d64bc6822cd2)
* Jun 1, 2001

[📄 Contents](http://www.informit.com/articles/article.aspx?p=167821&seqNum=4)

[␡](http://www.informit.com/articles/article.aspx?p=167821&seqNum=4)

1. [1 Oneway Messages](http://www.informit.com/articles/article.aspx?p=167821)
2. [2 Composing Oneway Messages](http://www.informit.com/articles/article.aspx?p=167821&seqNum=3)
3. 3 Services in Threads
4. [4 Parallel Decomposition](http://www.informit.com/articles/article.aspx?p=167821&seqNum=5)
5. [5 Active Objects](http://www.informit.com/articles/article.aspx?p=167821&seqNum=6)

* [⎙ Print](http://www.informit.com/articles/printerfriendly/167821)
* [+ Share This](http://www.addthis.com/bookmark.php)
* [💬 Discuss](http://www.informit.com/articles/article.aspx?p=167821&seqNum=4#articleDiscussion)

[< Back](http://www.informit.com/articles/article.aspx?p=167821&seqNum=3) **Page 3** of 5 [Next >](http://www.informit.com/articles/article.aspx?p=167821&seqNum=5)

## 4.3 Services in Threads

Many tasks compute results or provide services that are not immediately used by their clients, but are eventually required by them. In these situations, unlike those involving oneway messages, a client's actions at some point become dependent on completion of the task.

This section describes some of the available design alternatives: adding callbacks to oneway messages, relying on Thread.join, building utilities based on *Futures*, and creating worker threads. Section 4.4 revisits and extends these techniques in the context of improving the performance of computationally intensive tasks on parallel processors.

## 4.3.1 Completion Callbacks

From the perspective of pure oneway message passing, the most natural way to deal with completion is for a client to activate a task via a oneway message to a server, and for the server later to indicate completion by sending a oneway *callback* message to the caller. This efficient, asynchronous, notification-based style applies best in loosely-coupled designs in which completion of the service triggers some independent action in the client. Completion callback designs are sometimes structurally identical to Observer designs (see 3.5.2).

[**Figure 4-33**](javascript:popUp('/content/images/0201310090/elementLinks/04fig33.gif'))

For example, consider an application that offers several features, of which one or more require that a certain file be read in first. Because IO is relatively slow, you don't want to disable other features while the file is being read in — this would decrease responsiveness. One solution is to create a FileReaderservice that asynchronously reads in the file, and then issues a message back to the application when it has completed, so that the application can proceed with the feature(s) that require it.

### 4.3.1.1 Interfaces

To set up such a FileReader, or any other service using completion callbacks, you must first define a*client* interface for callback messages. The methods in this interface are substitutes of sorts for the kinds of return types and exceptions that would be associated with procedural versions of the service. This usually requires two kinds of methods, one associated with normal completion, and one associated with failure that is invoked upon any exception.

[**Figure 4-34**](javascript:popUp('/content/images/0201310090/elementLinks/04fig34.gif'))

Additionally, callback methods often require an argument indicating *which* action completed, so that the client can sort them out when there are multiple calls. In many cases this can be accomplished simply by sending back some of the call arguments. In more general schemes, the service hands back a unique identifier (usually known as a *cookie*) both as the return value for the initial request and as an argument in any callback method. Variants of this technique are used behind the scenes in remote invocation frameworks that implement procedural calls via asynchronous messages across networks.

In the case of FileReader, we could use interfaces such as:

interface FileReader {

void read(String filename, FileReaderClient client);

}

interface FileReaderClient {

void readCompleted(String filename, byte[ ] data);

void readFailed(String filename, IOException ex);

}

### 4.3.1.2 Implementations

There are two styles for implementing these interfaces, depending on whether you'd like the client or the server to create the thread that performs the service. Generally, if the service can be useful without running in its own thread, then control should be assigned to clients.

In the more typical case in which the use of threads is intrinsic to completion callback designs, control is assigned to the service method. Note that this causes callback methods to be executed in the thread constructed by the service, not one directly constructed by the client. This can lead to surprising results if any code relies on thread-specific properties such as ThreadLocal andjava.security.AccessControlContext (see 2.3.2) that are not known by the service.

Here we could implement a client and server using a service-creates-thread approach as:

class FileReaderApp implements FileReaderClient { // Fragments

protected FileReader reader = new AFileReader();

public void readCompleted(String filename, byte[ ] data) {

// ... use data ...

}

public void readFailed(String filename, IOException ex){

// ... deal with failure ...

}

public void actionRequiringFile() {

reader.read("AppFile", this);

}

public void actionNotRequiringFile() { ... }

}

class AFileReader implements FileReader {

public void read(final String fn, final FileReaderClient c) {

new Thread(new Runnable() {

public void run() { doRead(fn, c); }

}).start();

}

protected void doRead(String fn, FileReaderClient client) {

byte[ ] buffer = new byte[1024]; // just for illustration

try {

FileInputStream s = new FileInputStream(fn);

s.read(buffer);

if (client != null) client.readCompleted(fn, buffer);

}

catch (IOException ex) {

if (client != null) client.readFailed(fn, ex);

}

}

}

The service class here is written to deal with a null client argument, thus accommodating clients that do not need callbacks. While this is not particularly likely here, callbacks in many services can be treated as optional. As an alternative, you could define and use a NullFileReaderClient class that contains no-op versions of the callback methods (see Further Readings). Also, as far as the service is concerned, the callback target might as well be any object at all, for example some helper of the object that requests the service. You can also replace callbacks with event notifications using the techniques illustrated in 3.1.1.6.

### 4.3.1.3 Guarding callback methods

In some applications, clients can process completion callbacks only when they are in particular states. Here, the callback methods themselves should contain guards that suspend processing of each incoming callback until the client can deal with it.

For example, suppose we have a FileReaderClient that initiates a set of asynchronous file reads and needs to process them in the order issued. This construction mimics how remote invocations are usually handled: Typically each request is assigned a sequence number, and replies are processed in sequence order. This can be a risky strategy, since it will cause indefinite suspension of ready callbacks if one or more of them never completes. This drawback could be addressed by associating time-outs with the waits.

class FileApplication implements FileReaderClient { // Fragments

private String[ ] filenames;

private int currentCompletion; // index of ready file

public synchronized void readCompleted(String fn, byte[ ] d) {

// wait until ready to process this callback

while (!fn.equals(filenames[currentCompletion])) {

try { wait(); }

catch(InterruptedException ex) { return; }

}

// ... process data...

// wake up any other thread waiting on this condition:

++currentCompletion;

notifyAll();

}

public synchronized void readFailed(String fn, IOException e){

// similar...

}

public synchronized void readfiles() {

currentCompletion = 0;

for (int i = 0; i < filenames.length; ++i)

reader.read(filenames[i],this);

}

}

## 4.3.2 Joining Threads

While completion callbacks are very flexible, they are at best awkward to use when a caller just needs to wait out a particular task that it started.

[**Figure 4-35**](javascript:popUp('/content/images/0201310090/elementLinks/04fig35.gif'))

If an operation occurring in some thread A cannot continue until some thread B completes, you can block thread A via any of the waiting and notification techniques discussed in Chapter 3. For example, assuming the existence of a Latch (see 3.4.2) named terminated accessible from both threads A andB, thread A may wait via terminated.acquire(), and thread B may execute terminated.release()upon otherwise completing its task.

However, there is usually no reason to set up your own waiting and notification constructions, since this functionality is already provided by Thread.join: The join method blocks the caller while the targetisAlive. Terminating threads automatically perform notifications. The monitor object used internally for this waiting and notification is arbitrary and may vary across JVM implementations. In most, the target Thread object itself is used as the monitor object. (This is one reason for not extending classThread to add run methods that invoke waiting or notification methods.) In cases where these details of Thread.join don't fit the needs of a particular application, you can always fall back to the manual approach.

Either Thread.join or explicitly coded variants can be used in designs where a client needs a service to be performed but does not immediately rely on its results or effects. (This is sometimes known as*deferred-synchronous* invocation.) This is often the case when the service task is time-consuming and can benefit from CPU and/or IO parallelism, so that running it in a separate thread can improve overall throughput.

One common application is image processing. Obtaining the raw data for an image from a file or socket and then converting it into a form that can be displayed are time-consuming operations that involve both CPU and IO processing. Often, this processing can be overlapped with other display-related operations.

A version of this strategy is used by java.awt.MediaTracker and related classes, which should be used when they apply. Here, we'll illustrate a more generic version that can be extended and refined in various ways to support specialized applications.

To set this up, suppose there is a generic Pic interface for images, and a Renderer interface describing services that accept a URL pointing to image data and ultimately return a Pic. (In a more realistic setting, the render method would surely also throw various failure exceptions. Here, we will assume that it simply returns null on any failure.) Also, assume existence of a StandardRendererclass implementing interface Renderer.

Thread.join can be used to write clients such as the following PictureApp class (which invokes several made-up methods just for the sake of illustration). It creates a Runnable waiter object that both initiates the rendering thread and keeps track of the returned result.

[**Figure 4-36**](javascript:popUp('/content/images/0201310090/elementLinks/04fig36.gif'))

While it is common practice, the use of unsynchronized (or direct) access of internal result fields as seen in the waiter object is a bit delicate. Since access is not synchronized, correctness relies on the fact that both thread termination and the join method intrinsically employ synchronized methods or blocks (see 2.2.7).

interface Pic {

byte[ ] getImage();

}

interface Renderer {

Pic render(URL src);

}

class PictureApp { // Code sketch

// ...

private final Renderer renderer = new StandardRenderer();

public void show(final URL imageSource) {

class Waiter implements Runnable {

private Pic result = null;

Pic getResult() { return result; }

public void run() {

result = renderer.render(imageSource);

}

};

Waiter waiter = new Waiter();

Thread t = new Thread(waiter);

t.start();

displayBorders(); // do other things

displayCaption(); // while rendering

try {

t.join();

}

catch(InterruptedException e) {

cleanup();

return;

}

Pic pic = waiter.getResult();

if (pic != null)

displayImage(pic.getImage());

else

// ... deal with assumed rendering failure

}

}

Thread.join returns control to the caller whether the thread completed successfully or abnormally. For simplicity of illustration, nullness of the result field is used here to indicate any kind of failure, including cancellation of the renderer. The version in 4.3.3.1 illustrates a more responsible approach.

## 4.3.3 Futures

The operations underlying join-based constructions can be packaged in a more convenient and structured fashion by:

* Creating *Futures* — "virtual" data objects that automatically block when clients try to invoke their field accessors before their computation is complete. A Future acts as an "IOU" for a given data object.
* Creating versions of service methods that start up one or more threads and then return Future objects that are unblocked when computations complete.

Because the mechanics surrounding futures are built into data access and service methods, they can be applied in a general fashion only if both the data objects and the service methods are defined using interfaces, not classes. However, if the associated interfaces are defined, Futures are easy to set up. For example, a Future-based AsynchRenderer can employ proxies around concrete implementation classes (see 1.4.2):

[**Figure 4-37**](javascript:popUp('/content/images/0201310090/elementLinks/04fig37.gif'))

class AsynchRenderer implements Renderer {

private final Renderer renderer = new StandardRenderer();

static class FuturePic implements Pic { // inner class

private Pic pic = null;

private boolean ready = false;

synchronized void setPic(Pic p) {

pic = p;

ready = true;

notifyAll();

}

public synchronized byte[ ] getImage() {

while (!ready)

try { wait(); }

catch (InterruptedException e) { return null; }

return pic.getImage();

}

}

public Pic render(final URL src) {

final FuturePic p = new FuturePic();

new Thread(new Runnable() {

public void run() { p.setPic(renderer.render(src)); }

}).start();

return p;

}

}

For illustration, AsynchRenderer uses explicit waiting and notification operations based on a readyflag rather than relying on Thread.join.

Applications relying on this class can be written in a simple fashion:

class PicturAppWithFuture { // Code sketch

private final Renderer renderer = new AsynchRenderer();

public void show(final URL imageSource) {

Pic pic = renderer.render(imageSource);

displayBorders(); // do other things ...

displayCaption();

byte[ ] im = pic.getImage();

if (im != null)

displayImage(im);

else // deal with assumed rendering failure

}

}

### 4.3.3.1 Callables

Most designs based on Futures take exactly the form illustrated in class AsynchRenderer. The construction and use of such classes can be further standardized and automated by stepping up to a blander interface.

In the same way that interface Runnable describes any pure action, a Callable interface can be used to describe any service method that accepts an Object argument, returns an Object result, and may throw an Exception:

interface Callable {

Object call(Object arg) throws Exception;

}

The use of Object here (awkwardly) accommodates, for example, adaptation of methods accepting multiple arguments by stuffing them into array objects.

While there are other options, it is most convenient to package up support mechanics via a single class that coordinates usage. The following FutureResult class shows one set of choices. (It is a streamlined version of one in the util.concurrent package available from the online supplement.)

The FutureResult class maintains methods to get the result Object that is returned, or theException that is thrown by a Callable. Unlike our Pic versions where all failures were just indicated via null values, it deals with interruptions more honestly by throwing exceptions back to clients attempting to obtain results.

To differentiate properly between exceptions encountered in the service versus those encountered trying to execute the service, exceptions thrown by the Callable are repackaged using java.lang.reflect.InvocationTargetException, a general-purpose class for wrapping one exception inside another.

Also, for the sake of generality, the FutureResult does not itself create threads. Instead, it supports method setter that returns a Runnable that users can then execute within a thread or any other codeExecutor (see 4.1.4). This makes Callables usable within lightweight executable frameworks that are otherwise set up to handle tasks initiated via oneway messages. As an alternative strategy, you could set up a Caller framework that is otherwise similar to Executor, but is more specialized to the needs of service-style tasks, for example supporting methods to schedule execution, check status, and control responses to exceptions.

class FutureResult { // Fragments

protected Object value = null;

protected boolean ready = false;

protected InvocationTargetException exception = null;

public synchronized Object get()

throws InterruptedException, InvocationTargetException {

while (!ready) wait();

if (exception != null)

throw exception;

else

return value;

}

public Runnable setter(final Callable function) {

return new Runnable() {

public void run() {

try {

set(function.call());

}

catch(Throwable e) {

setException(e);

}

}

};

}

synchronized void set(Object result) {

value = result;

ready = true;

notifyAll();

}

synchronized void setException(Throwable e) {

exception = new InvocationTargetException(e);

ready = true;

notifyAll();

}

// ... other auxiliary and convenience methods ...

}

The FutureResult class can be used directly to support generic Futures or as a utility in constructing more specialized versions. As an example of direct use:

class PictureDisplayWithFutureResult { // Code sketch

private final Renderer renderer = new StandardRenderer();

// ...

public void show(final URL imageSource) {

try {

FutureResult futurePic = new FutureResult();

Runnable command = futurePic.setter(new Callable() {

public Object call() {

return renderer.render(imageSource);

}

});

new Thread(command).start();

displayBorders();

displayCaption();

displayImage(((Pic)(futurePic.get())).getImage());

}

catch (InterruptedException ex) {

cleanup();

return;

}

catch (InvocationTargetException ex) {

cleanup();

return;

}

}

}

This example demonstrates some of the minor awkwardnesses introduced by reliance on generic utilities that help standardize usage protocols. This is one reason that you might want to useFutureResult in turn to construct a more specialized and easier-to-use version with the same methods and structure as the AsynchRenderer class.

## 4.3.4 Scheduling Services

As discussed in 4.1.4, worker thread designs can sometimes improve performance compared to thread-per-task designs. They can also be used to schedule and optimize execution of service requests made by different clients.

As a famous example, consider a class controlling read and write access for a disk containing many cylinders but only one read/write head. The interface for the service contains just read and writemethods. In practice, it would surely use file block indicators instead of raw cylinder numbers and would deal with or throw various IO exceptions, here abbreviated as a single Failure exception.

interface Disk {

void read(int cylinderNumber, byte[ ] buffer) throws Failure;

void write(int cylinderNumber, byte[ ] buffer) throws Failure;

}

Rather than servicing access requests in the order they are made, it is faster on average to sweep the head across the cylinders, accessing cylinders in ascending order and then resetting the head position back to the beginning after each sweep. (Depending in part on the type of disk, it may be even better to arrange requests in both ascending and descending sweeps, but we will stick to this version.)

This policy would be tricky to implement without some kind of auxiliary data structure. The enabling condition for a request to execute is:

Wait until the current request cylinder number is the least greater cylinder number relative to that of the current disk head of all of those currently waiting, or is the least numbered cylinder if the head cylinder number is greater than that of all requests.

This condition is too awkward, inefficient, and possibly even deadlock-prone to try to coordinate across a set of otherwise independent clients. But it can be implemented fairly easily with the help of an ordered queue employed by a single worker thread. Tasks can be added to the queue in cylinder-based order, then executed when their turns arrive. This "elevator algorithm" is easiest to arrange by using a two-part queue, one for the current sweep and one for the next sweep.

[**Figure 4-38**](javascript:popUp('/content/images/0201310090/elementLinks/04fig38.gif'))

The resulting framework combines Future-like constructs with the worker thread designs from 4.1.4. To set this up, we can define a Runnable class to include the extra bookkeeping associated withDiskTasks. The queue class uses the semaphore-based approach discussed in 3.4.1, but here applied to ordered linked lists. The server class constructs a worker thread that runs tasks from the queue. The public service methods create tasks, place them on the queue, and then wait them out before returning to clients.

abstract class DiskTask implements Runnable {

protected final int cylinder; // read/write parameters

protected final byte[ ] buffer;

protected Failure exception = null; // to relay out

protected DiskTask next = null; // for use in queue

protected final Latch done = new Latch(); // status indicator

DiskTask(int c, byte[ ] b) { cylinder = c; buffer = b; }

abstract void access() throws Failure; // read or write

public void run() {

try { access(); }

catch (Failure ex) { setException(ex); }

finally { done.release(); }

}

void awaitCompletion() throws InterruptedException {

done.acquire();

}

synchronized Failure getException() { return exception; }

synchronized void setException(Failure f) { exception = f; }

}

class DiskReadTask extends DiskTask {

DiskReadTask(int c, byte[ ] b) { super(c, b); }

void access() throws Failure { /\* ... raw read ... \*/ }

}

class DiskWriteTask extends DiskTask {

DiskWriteTask(int c, byte[ ] b) { super(c, b); }

void access() throws Failure { /\* ... raw write ... \*/ }

}

class ScheduledDisk implements Disk {

protected final DiskTaskQueue tasks = new DiskTaskQueue();

public void read(int c, byte[ ] b) throws Failure {

readOrWrite(new DiskReadTask(c, b));

}

public void write(int c, byte[ ] b) throws Failure {

readOrWrite(new DiskWriteTask(c, b));

}

protected void readOrWrite(DiskTask t) throws Failure {

tasks.put(t);

try {

t.awaitCompletion();

}

catch (InterruptedException ex) {

Thread.currentThread().interrupt(); // propagate

throw new Failure(); // convert to failure exception

}

Failure f = t.getException();

if (f != null) throw f;

}

public ScheduledDisk() { // construct worker thread

new Thread(new Runnable() {

public void run() {

try {

for (;;) {

tasks.take().run();

}

}

catch (InterruptedException ie) {} // die

}

}).start();

}

}

class DiskTaskQueue {

protected DiskTask thisSweep = null;

protected DiskTask nextSweep = null;

protected int currentCylinder = 0;

protected final Semaphore available = new Semaphore(0);

void put(DiskTask t) {

insert(t);

available.release();

}

DiskTask take() throws InterruptedException {

available.acquire();

return extract();

}

synchronized void insert(DiskTask t) {

DiskTask q;

if (t.cylinder >= currentCylinder) { // determine queue

q = thisSweep;

if (q == null) { thisSweep = t; return; }

}

else {

q = nextSweep;

if (q == null) { nextSweep = t; return; }

}

DiskTask trail = q; // ordered linked list insert

q = trail.next;

for (;;) {

if (q == null || t.cylinder < q.cylinder) {

trail.next = t; t.next = q;

return;

}

else {

trail = q; q = q.next;

}

}

}

synchronized DiskTask extract() { // PRE: not empty

if (thisSweep == null) { // possibly swap queues

thisSweep = nextSweep;

nextSweep = null;

}

DiskTask t = thisSweep;

thisSweep = t.next;

currentCylinder = t.cylinder;

return t;

}

}

**4.4 Parallel Decomposition**

Parallel programs are specifically designed to take advantage of multiple CPUs for solving computation-intensive problems. The main performance goals are normally throughput and scalability — the number of computations that can be performed per unit time, and the potential for improvement when additional computational resources are available. However, these are often intertwined with other performance goals. For example, parallelism may also improve response latencies for a service that hands off work to a parallel execution facility.

Among the main challenges of parallelism in the Java programming language is to construct *portable*programs that can exploit multiple CPUs when they are present, while at the same time working well on single processors, as well as on time-shared multiprocessors that are often processing unrelated programs.

Some classic approaches to parallelism don't mesh well with these goals. Approaches that assume particular architectures, topologies, processor capabilities, or other fixed environmental constraints are ill suited to commonly available JVM implementations. While it is not a crime to build run-time systems with extensions specifically geared to particular parallel computers, and to write parallel programs specifically targeted to them, the associated programming techniques necessarily fall outside the scope of this book. Also, RMI and other distributed frameworks can be used to obtain parallelism across remote machines. In fact, most of the designs discussed here can be adapted to use serialization and remote invocation to achieve parallelism over local networks. This is becoming a common and efficient means of coarse-grained parallel processing. However, these mechanics also lie outside the scope of this book.

We instead focus on three families of task-based designs, fork/join parallelism, computation trees, and barriers. These techniques can yield very efficient programs that exploit multiple CPUs when present, yet still maintain portability and sequential efficiency. Empirically, they are known to scale well, at least up through dozens of CPUs. Moreover, even when these kinds of task-based parallel programs are tuned to maximally exploit a given hardware platform, they require only minor retunings to maximally exploit other platforms.

As of this writing, probably the most common targets for these techniques are applications servers and compute servers that are often, but by no means always, multiprocessors. In either case, we assume that CPU cycles are usually available, so the main goal is to exploit them to speed up the solution of computational problems. In other words, these techniques are unlikely to be very helpful when programs are run on computers that are already nearly saturated.

**4.4.1 Fork/Join**

Fork/join decomposition relies on parallel versions of divide-and-conquer techniques familiar in sequential algorithm design. Solutions take the form:

*pseudoclass* Solver { // Pseudocode

// ...

Result solve(Param problem) {

if (problem.size <= BASE\_CASE\_SIZE)

return directlySolve(problem);

else {

Result l, r;

*IN-PARALLEL* {

l = solve(lefthalf(problem));

r = solve(rightHalf(problem));

}

return combine(l, r);

}

}

}

It takes some hard work and inspiration to invent a divide-and-conquer algorithm. But many common computationally intensive problems have known solutions of approximately this form. Of course, there may be more than two recursive calls, multiple base cases, and arbitrary pre- and post-processing surrounding any of the cases.

Familiar sequential examples include quicksort, mergesort, and many data structure, matrix, and image processing algorithms. Sequential recursive divide-and-conquer designs are easy to parallelize when the recursive tasks are completely independent; that is, when they operate on different parts of a data set (for example different sections of an array) or solve different sub-problems, and need not otherwise communicate or coordinate actions. This often holds in recursive algorithms, even those not originally intended for parallel implementation.

Additionally, there are recursive versions of algorithms (for example, matrix multiplication) that are not used much in sequential contexts, but are more widely used on multiprocessors because of their readily parallelizable form. And other parallel algorithms perform extensive transformations and preprocessing to convert problems into a form that can be solved using fork/join techniques. (See Further Readings in 4.4.4.)

The *IN-PARALLEL* pseudocode is implemented by *forking* and later *joining* tasks performing the recursive calls. However, before discussing how to do this, we first examine issues and frameworks that permit efficient parallel execution of recursively generated tasks.

**4.4.1.1 Task granularity and structure**

Many of the design forces encountered when implementing fork/join designs surround task granularity:

**Maximizing parallelism.** In general, the smaller the tasks, the more opportunities for parallelism. All other things being equal, using many fine-grained tasks rather than only a few coarse-grained tasks keeps more CPUs busy, improves load balancing, locality and scalability, decreases the percentage of time that CPUs must idly wait for one another, and leads to greater throughput.

**Minimizing overhead.** Constructing and managing an object to process a task in parallel, rather than just invoking a method to process it serially, is the main unavoidable overhead associated with task-based programming compared with sequential solutions. It is intrinsically more costly to create and use task objects than to create and use stack-frames. Additionally, the use of task objects can add to the amount of argument and result data that must be transmitted and can impact garbage collection. All other things being equal, total overhead is minimized when there are only a few coarse-grained tasks.

**Minimizing contention.** A parallel decomposition is not going to lead to much speed-up if each task frequently communicates with others or must block waiting for resources held by others. Tasks should be of a size and structure that maintain as much independence as possible. They should minimize (in most cases, eliminate) use of shared resources, global (static) variables, locks, and other dependencies. Ideally, each task would contain simple straight-line code that runs to completion and then terminates. However, fork/join designs require at least some minimal synchronization. The main object that commences processing normally waits for all subtasks to finish before proceeding.

**Maximizing locality.** Each subtask should be the only one operating on some small piece of a problem, not only conceptually but also at the level of lower-level resources and memory access patterns. Refactorings that achieve good locality of reference can significantly improve performance on modern heavily cached processors. When dealing with large data sets, it is not uncommon to partition computations into subtasks with good locality even when parallelism is not the main goal. Recursive decomposition is often a productive way to achieve this. Parallelism accentuates the effects of locality. When parallel tasks all access different parts of a data set (for example, different regions of a common matrix), partitioning strategies that reduce the need to transmit updates across caches often achieve much better performance.

**4.4.1.2 Frameworks**

There is no general optimal solution to granularity and related task structuring issues. Any choice represents a compromise that best resolves the competing forces for the problem at hand. However, it is possible to build lightweight execution frameworks that support a wide range of choices along the continuum.

Thread objects are unnecessarily heavy vehicles for supporting purely computational fork/join tasks. For example, these tasks never need to block on IO, and never need to sleep. They require only an operation to synchronize across subtasks. Worker thread techniques discussed in 4.1.4 can be extended to construct frameworks efficiently supporting only the necessary constructs. While there are several approaches, for concreteness we'll limit discussion to a framework in util.concurrent that restricts all tasks to be subclasses of class FJTask. Here is a brief sketch of principal methods. More details are discussed along with examples in 4.4.1.4 through 4.4.1.7.

abstract class FJTask implements Runnable {

boolean isDone(); // True after task is run

void cancel(); // Prematurely set as done

void fork(); // Start a dependent task

void start(); // Start an arbitrary task

static void yield(); // Allow another task to run

void join(); // Yield caller until done

static void invoke(FJTask t); // Directly run t

static void coInvoke(FJTask t,

FJTask u); // Fork and join t and u

static void coInvoke(FJTask[ ] tasks); // coInvoke all

void reset(); // Clear to allow reuse

}

An associated FJTaskRunnerGroup class provides control and entry points into this framework. AFJTaskRunnerGroup is constructed with a given number of worker threads that should ordinarily be equal to the number of CPUs on a system. The class supports method invoke that starts up a main task, which will in turn normally create many others.

FJTasks must employ only these task control methods, not arbitrary Thread or monitor methods. While the names of these operations are the same or similar to those in class Thread, their implementations are very different. In particular, there are no general suspension facilities. For example, the joinoperation is implemented simply by having the underlying worker thread run other tasks to completion until the target task is noticed to have completed (via isDone). This wouldn't work at all with ordinary threads, but is effective and efficient when all tasks are structured as fork/join methods.

These kinds of trade-offs make FJTask construction and invocation substantially cheaper than would be possible for any class supporting the full Thread interface. As of this writing, on at least some platforms, the overhead of creating, running, and otherwise managing a FJTask for the kinds of examples illustrated below is only between four and ten times that of performing equivalent sequential method calls.

The main effect is to lessen the impact of overhead factors when making choices about task partitioning and granularity. The granularity threshold for using tasks can be fairly small — on the order of a few thousand instructions even in the most conservative cases — without noticeably degrading performance on uniprocessors. Programs can exploit as many CPUs as are available on even the largest platforms without the need for special tools to extract or manage parallelism. However, success also depends on construction of task classes and methods that themselves minimize overhead, avoid contention, and preserve locality.

**4.4.1.3 Defininhg tasks**

Sequential divide-and-conquer algorithms can be expressed as fork/join-based classes via the following steps:

1. Create a task class with:
   * Fields to hold arguments and results. Most should be strictly local to a task, never accessed from any other task. This eliminates the need for synchronization surrounding their use. However, in the typical case where result variables are accessed by other tasks, they should either be declared as volatile or be accessed only via synchronized methods.
   * A constructor that initializes argument variables.
   * A run method that executes the reworked method code.
2. Replace the original recursive case with code that:
   * Creates subtask objects.
   * Forks each one to run in parallel.
   * Joins each of them.
   * Combines results by accessing result variables in the subtask objects.
3. Replace (or extend) the original base case check with a *threshold* check. Problem sizes less than the threshold should use the original sequential code. This generalization of base case checks maintains efficiency when problem sizes are so small that task overhead overshadows potential gains from parallel execution. Tune performance by determining a good threshold size for the problem at hand.
4. Replace the original method with one that creates the associated task, waits it out, and returns any results. (In the FJTask framework, the outermost call is performed viaFJTaskRunnerGroup.invoke.)

**4.4.1.4 Fibonacci**

We'll illustrate the basic steps with a very boring and unrealistic, but very simple classic example: recursively computing *fib*, the Fibonacci function. This function can be programmed sequentially as:

int seqFib(int n) {

if (n <= 1)

return n;

else

return seqFib(n-1) + seqFib(n-2);

}

[**Figure 4-39**](javascript:popUp('/content/images/0201310090/elementLinks/04fig39.gif'))

This example is unrealistic because there is a much faster non-recursive solution for this particular problem, but it is a favorite for demonstrating both recursion and parallelism. Because it does so little other computation, it makes the basic structure of fork/join designs easier to see, yet it generates many recursive calls — at least *fib(n)* calls to compute *fib(n)*. The first few values of the sequence are 0, 1, 1, 2, 3, 5, 8; *fib(10)* is 55; *fib(20)* is 6,765; *fib(30)* is 832,040; *fib(40)* is 102,334,155.

Function seqFib can be transformed into a task class such as the following:

class Fib extends FJTask {

static final int sequentialThreshold = 13; // for tuning

volatile int number; // argument/result

Fib(int n) { number = n; }

int getAnswer() {

if (!isDone())

throw new IllegalStateException("Not yet computed");

return number;

}

public void run() {

int n = number;

if (n <= sequentialThreshold) // base case

number = seqFib(n);

else {

Fib f1 = new Fib(n - 1); // create subtasks

Fib f2 = new Fib(n - 2);

coInvoke(f1, f2); // fork then join both

number = f1.number + f2.number; // combine results

}

}

public static void main(String[ ] args) { // sample driver

try {

int groupSize = 2; // 2 worker threads

int num = 35; // compute fib(35)

FJTaskRunnerGroup group = new FJTaskRunnerGroup(groupSize);

Fib f = new Fib(num);

group.invoke(f);

int result = f.getAnswer();

System.out.println("Answer: " + result);

}

catch (InterruptedException ex) {} // die

}

}

**Notes:**

* The class maintains a field holding the argument for which to compute the Fibonacci function. Also, we need a variable to hold the result. However, as is fairly typical in such classes, there is no need to keep two variables. For economy (bearing in mind that many millions of Fib objects might be constructed in the course of a computation), we can micro-optimize to use one variable, and overwrite the argument with its result after it is computed. (This is the first of several hand-optimizations that are uncomfortably petty, but are shown here in order to demonstrate minor tweaks that can be pragmatically important in constructing efficient parallel programs.)
* The number field is declared as volatile to ensure visibility from other tasks/threads after it is computed (see 2.2.7). Here and in subsequent examples, volatile fields are read and/or written only once per task execution, and otherwise held in local variables. This avoids interfering with potential compiler optimizations that are otherwise disabled when using volatile.
* Alternatively, we could have synchronized access to the number field. But there is no good reason to do so. The use of volatile fields is much more common in lightweight parallel task frameworks than in general-purpose concurrent programming. Tasks usually do not require other synchronization and control mechanics, yet often need to communicate results via field access. The most common reason for using synchronized instead of volatile is to deal with arrays. Individual array elements cannot be declared as volatile. Processing arrays within synchronized methods or blocks is the simplest way to ensure visibility of array updates, even in the typical case in which locking is not otherwise required. An occasionally attractive alternative is instead to create arrays each of whose elements is a forwarding object with volatile fields.
* The method isDone returns true after the completion of a run method that has been executed via invoke or coInvoke. It is used as a guard in getAnswer to help detect programming errors that could occur if the ultimate consumer of an answer tries to access it prematurely. (There is no chance of this happening here, but this safeguard helps avoid unintended usages.)
* The sequentialThreshold constant establishes granularity. It represents the balance point at which it is not worth the overhead to create tasks, also reflecting the goal of maintaining good sequential performance. For example, on one set of runs on a four-CPU system, setting sequentialThreshold to 13 resulted in a 4% performance degradation versusseqFib for large argument values when using a single worker thread. But it sped up by a factor of at least 3.8 with four worker threads, processing several million Fib tasks per second.
* Rather than wiring in a compile-time constant, we could have defined the threshold as a run-time variable and set it to a value based on the number of CPUs available or other platform characteristics. This is useful in task-based programs that do not scale linearly, as is likely to be true even here. As the number of CPUs increase, so do communication and resource management costs, which could be balanced by increasing the threshold.
* The parallel analog of recursion is performed via a convenient method, coInvoke(FJTask t, FJTask u), which in turn acts as: t.fork(); invoke(u); t.join();
* The fork method is a specialized analog of Thread.start. A forked task is always processed in stack-based LIFO order when it is run by the same underlying worker thread that spawned it, but in queue-based FIFO order with respect to other tasks if run by another worker thread running in parallel. This represents a cross of sorts between normal stack-based sequential calls, and normal queue-based thread scheduling. This policy (implemented via double-ended scheduling queues) is ideal for recursive task-based parallelism (see Further Readings), and more generally whenever dealing with strictly dependent tasks — those that are spawned either by the tasks that ultimately join them or by their subtasks.
* In contrast, FJTask.start behaves more like Thread.start. It employs queue-based FIFO scheduling with respect to all worker threads. It is used, for example, byFJTaskRunnerGroup.invoke to start up execution of a new main task.
* The join method should be used only for tasks initiated via fork. It exploits termination dependency patterns of fork/join subtasks to optimize execution.
* The FJTask.invoke method runs the body of one task within another task, and waits out completion. Seen differently, it is the one-task version of coInvoke, an optimization ofu.fork(); u.join().

Effective use of any lightweight executable framework requires the same understanding of support methods and their semantics as does programming with ordinary Threads. The FJTask framework exploits the symbiosis between recursion and parallel decomposition, and so encourages the divide-and-conquer programming style seen in Fib. However, the range of programming idioms and design patterns conforming to this general style is fairly broad, as illustrated by the following examples.

**4.4.1.5 Linking subtasks**

Fork/join techniques may be applied even when the number of forked subtasks varies dynamically. Among several related tactics for carrying this out, you can add link fields so that subtasks can be maintained in lists. After spawning all tasks, an accumulate (also known as *reduction*) operation can traverse the list sequentially, joining and using the results of each subtask.

Stretching the Fib example a bit, the FibVL class illustrates one way to set this up. This style of solution is not especially useful here, but is applicable in contexts in which a dynamic number of subtasks are created, possibly across different methods. Notice that the subtasks here are joined in the opposite order in which they were created. Since the processing order of results does not matter here, we use the simplest possible linking algorithm (prepending), which happens to reverse the order of tasks during traversal. This strategy applies whenever the accumulation step is commutative and associative with respect to results, so tasks can be processed in any order. If the order did matter, we would need to adjust list construction or traversal accordingly.

class FibVL extends FJTask {

volatile int number; // as before

final FibVL next; // embedded linked list of sibling tasks

FibVL(int n, FibVL list) { number = n; next = list; }

public void run() {

int n = number;

if (n <= sequentialThreshold)

number = seqFib(n);

else {

FibVL forked = null; // list of subtasks

forked = new FibVL(n - 1, forked); // prepends to list

forked.fork();

forked = new FibVL(n - 2, forked);

forked.fork();

number = accumulate(forked);

}

}

// Traverse list, joining each subtask and adding to result

int accumulate(FibVL list) {

int sum = 0;

for (FibVL f = list; f != null; f = f.next) {

f.join();

sum += f.number;

}

return sum;

}

}

**4.4.1.6 Callbacks**

Recursive task-based fork/join parallelism may be extended to apply when other local synchronization conditions are used instead of join. In the FJTask framework, t.join() is implemented as an optimized version of:

while (!t.isDone()) yield();

Method yield here allows the underlying worker thread to process other tasks. (More specifically, in theFJTask framework, the thread *will* process at least one other task if one exists.)

Any other condition may be used in this construction rather than isDone, as long as you are certain that the predicate being waited for will eventually become true due to the actions of a subtask (or one of its subtasks, and so on). For example, rather than relying on join, task control can rely on counters that keep track of task creation and completion. A counter can be incremented on each fork and decremented when the forked task has produced a result. This and related counter-based schemes can be attractive choices when subtasks communicate back results via callbacks rather than via access to result fields. Counters of this form are small-scale, localized versions of the barriers discussed in 4.4.3.

Callback-based fork/join designs are seen, for example, in problem-solving algorithms, games, searching, and logic programming. In many such applications, the number of subtasks that are forked can vary dynamically, and subtask results are better captured by method calls than by field extraction.

Callback-based approaches also permit greater asynchrony than techniques such as the linked tasks in 4.4.1.5. This can lead to better performance when subtasks differ in expected duration, since the result processing of quickly completing subtasks can sometimes overlap with continued processing of longer ones. However, this design gives up all result ordering guarantees, and thus is applicable only when subtask result processing is completely independent of the order in which results are produced.

Callback counters are used in the following class FibVCB, which is not at all well-suited for the problem at hand but serves to exemplify techniques. This code illustrates a typical but delicate combination of task-local variables, volatiles, and locking in an effort to keep task control overhead to a minimum:

class FibVCB extends FJTask {

// ...

volatile int number = 0; // as before

final FibVCB parent; // is null for outermost call

int callbacksExpected = 0;

volatile int callbacksReceived = 0;

FibVCB(int n, FibVCB p) { number = n; parent = p; }

// Callback method invoked by subtasks upon completion

synchronized void addToResult(int n) {

number += n;

++callbacksReceived;

}

public void run() { // same structure as join-based version

int n = number;

if (n <= sequentialThreshold)

number = seqFib(n);

else {

// Clear number so subtasks can fill in

number = 0;

// Establish number of callbacks expected

callbacksExpected = 2;

new FibVCB(n - 1, this).fork();

new FibVCB(n - 2, this).fork();

// Wait for callbacks from children

while (callbacksReceived < callbacksExpected) yield();

}

// Call back parent

if (parent != null) parent.addToResult(number);

}

}

**Notes:**

* All mutual exclusion locking is restricted to small code segments protecting field accesses, as must be true for any class in a lightweight task framework. Tasks are not allowed to block unless they are sure they will be able to continue soon. In particular, this framework unenforceably requires that synchronized blocks *not* span forks and subsequent joins oryields.
* To help eliminate some synchronization, the callback count is split into two counters,callbacksExpected and callbacksReceived. The task is done when they are equal.
* The callbacksExpected counter is used only by the current task, so access need not besynchronized, and it need not be volatile. In fact, since exactly two callbacks are always expected in the recursive case and the value is never needed outside the run method, this class could easily be reworked in a way that eliminates all need for this variable. However, such a variable is needed in more typical callback-based designs where the number of forks may vary dynamically and may be generated across multiple methods.
* The addToResult callback method must be synchronized to avoid interference problems when subtasks call back at about the same time.
* So long as both number and callbacksReceived are declared as volatile, andcallbacksReceived is updated as the last statement of addToResult, the yield loop test need not involve synchronization because it is waiting for a latching threshold that, once reached, will never change (see 3.4.2.1).
* We could also define a reworked getAnswer method that uses these mechanics so that it returns an answer if all callbacks have been received. However, since this method is designed to be called by external (non-task) clients upon completion of the overall computation, there is no compelling reason to do this. The version from the original Fibclass suffices.
* Despite these measures, the overhead associated with task control in this version is greater than that of the original version using coInvoke. If you were to use it anyway, you would probably choose a slightly larger sequential threshold, and thus exploit slightly less parallelism.

**4.4.1.7 Cancellation**

In some designs, there is no need for keeping counts of callbacks or exhaustively traversing through subtask lists. Instead, tasks complete when any subtask (or one of its subtasks, and so on) arrives at a suitable result. In these cases, you can avoid wasting computation by cancelling any subtasks in the midst of producing results that will not be needed.

The options here are similar to those seen in other situations involving cancellation (see 3.1.2). For example, subtasks can regularly invoke a method (perhaps isDone) in their parents that indicates that an answer has already been found, and if so to return early. They must also set their own status, so any of their subtasks can do the same. This can be implemented here using FJTask.cancel that just prematurely sets isDone status. This suppresses execution of tasks that have not yet been started, but has no effect on tasks in the midst of execution unless the tasks' run methods themselves detect updated status and deal with it.

When an entire set of tasks are all trying to compute a single result, an even simpler strategy suffices: Tasks may regularly check a global (static) variable that indicates completion. However, when there are many tasks, and many CPUs, more localized strategies may still be preferable to one that places so much pressure on the underlying system by generating many accesses to the same memory location, especially if it must be accessed under synchronization. Additionally, bear in mind that the total overhead associated with cancellation should be less than the cost of just letting small tasks run even if their results are not needed.

For example, here is a class that solves the classic N-Queens problem, searching for the placement of N queens that do not attack each other on a chessboard of size NxN. For simplicity of illustration, it relies on a static Result variable. Here tasks check for cancellation only upon entry into the method. They will continue looping through possible extensions even if a result has already been found. However, the generated tasks will immediately exit. This can be slightly wasteful, but may obtain a solution more quickly than a version that checks for completion upon every iteration of every task.

Note also here that the tasks do not bother joining their subtasks since there is no reason to do so. Only the ultimate external caller (in main) needs to wait for a solution; this is supported here by adding standard waiting and notification methods to the Result class. (Also, for compactness, this version does not employ any kind of granularity threshold. It is easy to add one, for example by directly exploring moves rather than forking subtasks when the number of rows is close to the board size.)

class NQueens extends FJTask {

static int boardSize; // fixed after initialization in main

// Boards are arrays where each cell represents a row,

// and holds the column number of the queen in that row

static class Result { // holder for ultimate result

private int[ ] board = null; // non-null when solved

synchronized boolean solved() { return board != null; }

synchronized void set(int[ ] b) { // Support use by non-Tasks

if (board == null) { board = b; notifyAll(); }

}

synchronized int[ ] await() throws InterruptedException {

while (board == null) wait();

return board;

}

}

static final Result result = new Result();

public static void main(String[ ] args) {

boardSize = ...;

FJTaskRunnerGroup tasks = new FJTaskRunnerGroup(...);

int[ ] initialBoard = new int[0]; // start with empty board

tasks.execute(new NQueens(initialBoard));

int[ ] board = result.await();

// ...

}

final int[ ] sofar; // initial configuration

NQueens(int[ ] board) { this.sofar = board; }

public void run() {

if (!result.solved()) { // skip if already solved

int row = sofar.length;

if (row >= boardSize) // done

result.set(sofar);

else { // try all expansions

for (int q = 0; q < boardSize; ++q) {

// Check if queen can be placed in column q of next row

boolean attacked = false;

for (int i = 0; i < row; ++i) {

int p = sofar[i];

if (q == p || q == p - (row-i) || q == p + (row-i)) {

attacked = true;

break;

}

}

// If so, fork to explore moves from new configuration

if (!attacked) {

// build extended board representation

int[ ] next = new int[row+1];

for (int k = 0; k < row; ++k) next[k] = sofar[k];

next[row] = q;

new NQueens(next).fork();

}

}

}

}

}

}

**4.4.2 Computation Trees**

A number of computationally intensive algorithms involve tasks of the form:

For a fixed number of steps, or until convergence, do {

Update one section of a problem;

Wait for other tasks to finish updating their sections;

}

[**Figure 4-40**](javascript:popUp('/content/images/0201310090/elementLinks/04fig40.gif'))

Most often, such algorithms perform update operations on partitioned arrays, matrices, or image representations. For example, many physical dynamics problems involve repeated local updates to the cells of a matrix. *Jacobi* algorithms and related relaxation techniques repeatedly recalculate estimated values across neighboring cells, typically using an averaging formula such as:

void oneStep(double[ ][ ] oldMatrix,

double[ ][ ] newMatrix, int i, int j) {

newMatrix[i][j] = 0.25 \* (oldMatrix[i-1][j] +

oldMatrix[i][j-1] +

oldMatrix[i+1][j] +

oldMatrix[i][j+1]);

}

Normally, to save space, two different matrices are swapped as newMatrix and oldMatrix across successive steps.

Algorithms requiring that *all* tasks periodically wait for *all* others to complete do not always scale quite as well as more loosely coupled fork/join designs. Even so, these algorithms are common, efficient, and amenable to significant parallel speedups.

**4.4.2.1 Building and using trees**

It would be inefficient to repeatedly apply fork/join decomposition in iterative designs in order to update sections in parallel. Because the sections are the same across iterations, they can be constructed just once and then repeatedly invoked so that on each iteration, the corresponding updates execute in the same order as would be produced by a recursive solution.

Computation trees are explicit representations of the tree-structured computations implicitly arising in fork/join recursion. These trees have two kinds of nodes, internal nodes and leaf nodes, corresponding to the recursive and base cases of a recursive solution. They can be constructed and used for iterative update problems via the following steps:

1. Create a tree of task objects representing the recursive partitions, where:
   * Each internal node contains references to subpartitions, and has an update method that performs fork/join processing of each of them.
   * Each leaf node represents a finest-granularity partition, and has an update method that operates directly on it.
2. For a fixed number of steps, or until convergence, do:
   * Execute the task performing the root partition's update method.

For example, the following code illustrates the highlights of a set of classes that perform Jacobi iteration using the averaging formula shown above. In addition to updating, this version also keeps track of the differences among computed cell values across iterations, and stops when the maximum difference is within a constant EPSILON. Also, like many programs of this form, this code assumes that the matrices have been set up with extra edge cells that are never updated, so boundary conditions never need to be checked. (Alternatives include recomputing edge values using special edge formulas after each pass, and treating edges as toroidally wrapping around the mesh.)

The recursive decomposition strategy used here is to divide the mesh into quadrants, stopping when the number of cells is at most leafCells, which serves as the granularity threshold. This strategy works well so long as the numbers of rows and columns in the matrix are approximately equal. If they are not, additional classes and methods could be defined to divide across only one dimension at a time. The approach here assumes that the matrix as a whole already exists, so rather than actually dividing up cells, task nodes just keep track of the row and column offsets of this matrix that each partition is working on.

The subclass-based design used here reflects the different structure and behavior of internal versus leaf nodes. Both are subclasses of abstract base JTree:

abstract class JTree extends FJTask {

volatile double maxDiff; // for convergence check

}

class Interior extends JTree {

private final JTree[ ] quads;

Interior(JTree q1, JTree q2, JTree q3, JTree q4) {

quads = new JTree[ ] { q1, q2, q3, q4 };

}

public void run() {

coInvoke(quads);

double md = 0.0;

for (int i = 0; i < quads.length; ++i) {

md = Math.max(md,quads[i].maxDiff);

quads[i].reset();

}

maxDiff = md;

}

}

class Leaf extends JTree {

private final double[ ][ ] A; private final double[ ][ ] B;

private final int loRow; private final int hiRow;

private final int loCol; private final int hiCol;

private int steps = 0;

Leaf(double[ ][ ] A, double[ ][ ] B,

int loRow, int hiRow, int loCol, int hiCol) {

this.A = A; this.B = B;

this.loRow = loRow; this.hiRow = hiRow;

this.loCol = loCol; this.hiCol = hiCol;

}

public synchronized void run() {

boolean AtoB = (steps++ % 2) == 0;

double[ ][ ] a = (AtoB)? A : B;

double[ ][ ] b = (AtoB)? B : A;

double md = 0.0;

for (int i = loRow; i <= hiRow; ++i) {

for (int j = loCol; j <= hiCol; ++j) {

b[i][j] = 0.25 \* (a[i-1][j] + a[i][j-1] +

a[i+1][j] + a[i][j+1]);

md = Math.max(md, Math.abs(b[i][j] - a[i][j]));

}

}

maxDiff = md;

}

}

The driver class first builds a tree that represents the partitioning of its argument matrix. The build method could itself be parallelized. But because the base actions are just node constructions, the granularity threshold would be so high that parallelization would be worthwhile only for huge problem sizes.

The run method repeatedly sets the root task in motion and waits out completion. For simplicity of illustration, it continues until convergence. Among other changes necessary to turn this into a realistic program, you would need to initialize the matrices and deal with possible lack of convergence within a bounded number of iterations. Because each iteration entails a full synchronization point waiting for the root task to finish, it is relatively simple to insert additional operations that maintain or report global status between iterations.

class Jacobi extends FJTask {

static final double EPSILON = 0.001; // convergence criterion

final JTree root;

final int maxSteps;

Jacobi(double[ ][ ] A, double[][] B,

int firstRow, int lastRow, int firstCol, int lastCol,

int maxSteps, int leafCells) {

this.maxSteps = maxSteps;

root = build(A, B, firstRow, lastRow, firstCol, lastCol,

leafCells);

}

public void run() {

for (int i = 0; i < maxSteps; ++i) {

invoke(root);

if (root.maxDiff < EPSILON) {

System.out.println("Converged");

return;

}

else root.reset();

}

}

static JTree build(double[ ][ ] a, double[ ][ ] b,

int lr, int hr, int lc, int hc, int size) {

if ((hr - lr + 1) \* (hc - lc + 1) <= size)

return new Leaf(a, b, lr, hr, lc, hc);

int mr = (lr + hr) / 2; // midpoints

int mc = (lc + hc) / 2;

return new Interior(build(a, b, lr, mr, lc, mc, size),

build(a, b, lr, mr, mc+1, hc, size),

build(a, b, mr+1, hr, lc, mc, size),

build(a, b, mr+1, hr, mc+1, hc, size));

}

}

**4.4.3 Barriers**

Recursive decomposition is a powerful and flexible technique, but does not always fit well with the structure of iterative problems, and usually requires adoption of a lightweight execution framework for efficient implementation. A more direct path to a solution of many iterative problems is first to divide the problem into *segments*, each with an associated task performing a loop that must periodically wait for other segments to complete. From the perspective of tree-based approaches, these designs flatten out all the internal nodes and just deal with the leaves.

[**Figure 4-41**](javascript:popUp('/content/images/0201310090/elementLinks/04fig41.gif'))

As with recursive tasks, there are opportunities to specialize Threads to make them more attuned to the demands of parallel iteration (see Further Readings). However, there is usually less to be gained by doing so, in part because all thread construction overhead is restricted to the start-up phase. Here we illustrate the basic mechanics using regular Threads each executing a single Runnable. When usingThreads, granularity thresholds must in general be substantially higher than when using lightweight executable classes (although still substantially lower than those needed in distributed parallel designs). But the basic logic of iterative algorithms is otherwise identical, regardless of granularity. In many iterative problems, little potential parallelism is wasted by using coarse granularities. When all threads perform approximately the same actions for approximately the same durations, creating only as many tasks as CPUs, or perhaps a small multiple of the number of CPUs, can work well.

While it is always possible to hand-craft the necessary control mechanics using waiting and notification constructs, it is both more convenient and less error-prone instead to rely on standardized synchronization aids that encapsulate these mechanics. The synchronization device of choice in iterative designs is a *cyclic barrier*. A cyclic barrier is initialized with a fixed number of parties that will be repeatedly synchronizing. It supports only one method, barrier, that forces each caller to wait until all parties have invoked the method, and then resets for the next iteration. A basic CyclicBarrier class can be defined as follows:

class CyclicBarrier {

protected final int parties;

protected int count; // parties currently being waited for

protected int resets = 0; // times barrier has been tripped

CyclicBarrier(int c) { count = parties = c; }

synchronized int barrier() throws InterruptedException {

int index = --count;

if (index > 0) { // not yet tripped

int r = resets; // wait until next reset

do { wait(); } while (resets == r);

}

else { // trip

count = parties; // reset count for next time

++resets;

notifyAll(); // cause all other parties to resume

}

return index;

}

}

(The util.concurrent version of this class available from the online supplement deals more responsibly with interruptions and time-outs. Fancier versions that reduce memory contention on the lock and on the fields may be worth constructing on systems with very large numbers of processors.)

The CyclicBarrier.barrier method defined here returns the number of other threads that were still waiting when the barrier was entered, which can be useful in some algorithms. As another by-product, the barrier method is intrinsically synchronized, so it also serves as a memory barrier to ensure flushes and loads of array element values in its most typical usage contexts (see 2.2.7).

A barrier may also be construed as a simple consensus operator (see 3.6). It gathers "votes" among several threads about whether they should all continue to the next iteration. Release occurs when all votes have been collected and agreement has thus been reached. (However, unlike transaction frameworks, threads using this CyclicBarrier class are not allowed to vote "no".)

With barriers, many parallel iterative algorithms become easy to express. In the simplest cases, these programs might take the form (eliding all problem-specific details):

class Segment implements Runnable { // Code sketch

final CyclicBarrier bar; // shared by all segments

Segment(CyclicBarrier b, ...) { bar = b; ...; }

void update() { ... }

public void run() {

// ...

for (int i = 0; i < iterations; ++i) {

update();

bar.barrier();

}

// ...

}

}

class Driver {

// ...

void compute(Problem problem) throws ... {

int n = problem.size / granularity;

CyclicBarrier barrier = new CyclicBarrier(n);

Thread[ ] threads = new Thread[n];

// create

for (int i = 0; i < n; ++i)

threads[i] = new Thread(new Segment(barrier, ...));

// trigger

for (int i = 0; i < n; ++i) threads[i].start();

// await termination

for (int i = 0; i < n; ++i) threads[i].join();

}

}

This structure suffices for problems requiring known numbers of iterations. However, many problems require checks for convergence or some other global property between iterations. (Conversely, in a few*chaotic* relaxation algorithms you don't even need a barrier after each iteration, but can instead let segments free-run for a while between barriers and/or checks.)

One way to provide convergence checks is to rework the CyclicBarrier class to optionally run a supplied Runnable command whenever a barrier is about to be reset. A more classic approach, which illustrates a technique useful in other contexts as well, is to rely on the index returned by barrier. The caller obtaining index zero (as an arbitrary, but always legal choice) can perform the check while all others are quietly waiting for a *second* barrier.

For example, here a a barrier-based version of a segment class for the Jacobi problem described in 4.4.2. Collections of JacobiSegment objects can be initialized and run by a driver of the generic form given above.

class JacobiSegment implements Runnable { // Incomplete

// These are same as in Leaf class version:

double[ ][ ] A; double[ ][ ] B;

final int firstRow; final int lastRow;

final int firstCol; final int lastCol;

volatile double maxDiff;

int steps = 0;

void update() { /\* Nearly same as Leaf.run \*/ }

final CyclicBarrier bar;

final JacobiSegment[ ] allSegments; // for convergence check

volatile boolean converged = false;

JacobiSegment(double[ ][ ] A, double[ ][ ] B,

int firstRow, int lastRow,

int firstCol, int lastCol,

CyclicBarrier b, JacobiSegment[ ] allSegments) {

this.A = A; this.B = B;

this.firstRow = firstRow; this.lastRow = lastRow;

this.firstCol = firstCol; this.lastCol = lastCol;

this.bar = b;

this.allSegments = allSegments;

}

public void run() {

try {

while (!converged) {

update();

int myIndex = bar.barrier(); // wait for all to update

if (myIndex == 0) convergenceCheck();

bar.barrier(); // wait for convergence check

}

}

catch(Exception ex) {

// clean up ...

}

}

void convergenceCheck() {

for (int i = 0; i < allSegments.length; ++i)

if (allSegments[i].maxDiff > EPSILON) return;

for (int i = 0; i < allSegments.length; ++i)

allSegments[i].converged = true;

}

}

**4.4.4 Further Readings**

For a survey of approaches to high-performance parallel processing, see

Skillicorn, David, and Domenico Talia, "Models and Languages for Parallel Computation",*Computing Surveys*, June 1998.

Most texts on parallel programming concentrate on algorithms designed for use on fine-grained parallel machine architectures, but also cover design techniques and algorithms that can be implemented using the kinds of stock multiprocessors most amenable to supporting a JVM. See, for example:

Foster, Ian. *Designing and Building Parallel Programs*, Addison Wesley, 1995.

Roosta, Seyed. *Parallel Processing and Parallel Algorithms*, Springer-Verlag, 1999.

Wilson, Gregory. *Practical Parallel Programming*, MIT Press, 1995.

Zomaya, Albert (ed.). *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996.

Pattern-based accounts of parallel programming include:

Massingill, Berna, Timothy Mattson, and Beverly Sanders. *A Pattern Language for Parallel Application Programming*, Technical report, University of Florida, 1999.

MacDonald, Steve, Duane Szafron, and Jonathan Schaeffer. "Object-Oriented Pattern-Based Parallel Programming with Automatically Generated Frameworks", in *Proceedings of the 5th USENIX Conference on Object-Oriented Tools and Systems (COOTS)*, 1999.

The FJTask framework internally relies on a *work-stealing* task scheduler based on the one in Cilk, a C-based parallel programming framework. In work-stealing schedulers, each worker thread normally runs (in LIFO order) tasks that it constructs, but when idle steals (in FIFO order) those constructed by other worker threads. More details, including explanations of the senses in which this scheme is optimal for recursive fork/join programs, may be found in:

Frigo, Matteo, Charles Leiserson, and Keith Randall. "The Implementation of the Cilk-5 Multithreaded Language", *Proceedings of 998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1998.

The online supplement includes more realistic examples of the techniques discussed in this section. It also provides links to the Cilk package and related frameworks, including Hood (a C++ follow-on to Cilk) and Filaments (a C package that includes a specialized framework supporting barrier-based iterative computation).

[< Back](http://www.informit.com/articles/article.aspx?p=167821&seqNum=4) **Page 4** of 5 [Next >](http://www.informit.com/articles/article.aspx?p=167821&seqNum=6)

* [+ Share This](http://www.addthis.com/bookmark.php)
* [🔖 Save To Your Account](http://www.informit.com/articles/article.aspx?p=167821&seqNum=5#addToWishList)

[Home](http://www.informit.com/) > [Articles](http://www.informit.com/articles/index.aspx) > [Programming](http://www.informit.com/articles/index.aspx?st=60206) > [Java](http://www.informit.com/articles/index.aspx?st=60209)

## 4.5 Active Objects

In the task-based frameworks illustrated throughout most of this chapter, threads are used to propel conceptually active messages sent among conceptually passive objects. However, it can be productive to approach some design problems from the opposite perspective — active objects sending each other passive messages.

To illustrate, consider an active object that conforms to the WaterTank description in Chapter 1:

*pseudoclass* ActiveWaterTank extends Thread { // Pseudocode

// ...

public void run() {

for (;;) {

*accept* message;

if (message *is of form* addWater(float amount)) {

if (currentVolume >= capacity) {

if (overflow != null) {

*send* overflow.addWater(amount);

*accept* response;

if (response *is of form* OverflowException)

*reply* response;

else ...

else ...

else ...

}

else if (message *is of form* removeWater(float amount)) {

...

}

}

}

}

Pseudocode is used here because there is no built-in syntax for passing messages from one active object to another, only for direct invocation among passive objects. However, as discussed in 4.1.1, similar issues may be encountered even when using passive objects. Any of the solutions described there apply equally well here: adopting message formats of various kinds, transported across streams, channels, event queues, pipes, sockets, and so on. In fact, as shown in the WebService example leading off this chapter, it is easy to add task-based constructions to designs otherwise based on active objects. Conversely, most task-based designs discussed in this chapter work equally well when some objects are active rather than passive.

[**Figure 4-42**](javascript:popUp('graphics/04fig42.gif'))

Further, the use of Runnables as messages leads to a boring but universal (at least in some senses) form of active object: a minor variant of a common worker thread design that also conforms to the initial abstract characterization of active objects as interpreters in 1.2.4:

class ActiveRunnableExecutor extends Thread {

Channel me = ... // used for all incoming messages

public void run() {

try {

for (;;) {

((Runnable)(me.take())).run();

}

}

catch (InterruptedException ie) {} // die

}

}

Of course, such classes are not very useful unless they also include internal methods that manufactureRunnables to execute and/or send to other active objects. It is possible, but unnatural, to write entire programs in this fashion.

However, many components in reactive systems can be usefully construed as active objects that operate under more constrained rules and message-passing disciplines. This includes especially those objects that interact with other computers or devices, often the main externally visible objects in a program.

In distributed frameworks such as CORBA and RMI, externally visible active objects are themselves ascribed interfaces listing the messages that they accept. Internally, they usually have a more uniform structure than does ActiveWaterTank. Typically, they contain a main run loop that repeatedly accepts external requests, dispatches to internal passive objects providing the corresponding service, and then constructs reply messages that are sent back to clients. (The internal passive objects are the ones explicitly programmed when using CORBA and RMI. The active objects, sometimes known as*skeletons*, are usually generated automatically by tools.)

It is very possible to take an active, actor-style approach to the design of other components as well. One reason for designing entire systems from this point of view is to take advantage of well-developed theory and design techniques associated with particular sets of rules surrounding active entities and their messages. The remainder of this section gives a brief overview of the most well-known and influential such framework, CSP.

## 4.5.1 CSP

C.A.R. Hoare's theory of Communicating Sequential Processes (CSP) provides both a formal approach to concurrency and an associated set of design techniques. As discussed in the Further Readings in 4.5.2, there are a number of closely related approaches, but CSP has had the largest impact on concurrent design and programming. CSP has served as the basis of programming languages (includingoccam), was influential in the design of others (including Ada), and can be supported in the Java programming language through the use of library classes.

The following account illustrates the JCSP package developed by Peter Welch and colleagues. The package is available via links from the online supplement. This section provides only a brief synopsis. Interested readers will want to obtain copies of the package, its documentation, and related texts.

### 4.5.1.1 Processes and channels

A CSP *process* can be construed as a special kind of actor-style object, in which:

* Processes have no method interface and no externally invocable methods. Because there are no invocable methods, it is impossible for methods to be invoked by different threads. Thus there is no need for explicit locking.
* Processes communicate only by reading and writing data across *channels*.
* Processes have no identity, and so cannot be explicitly referenced. However, channels serve as analogs of references (see 1.2.4), allowing communication with whichever process is at the other end of a channel.
* Processes need not spin forever in a loop accepting messages (although many do). They may read and write messages on various channels as desired.

A CSP *channel* can be construed as a special kind of Channel, in which:

* All channels are synchronous (see 3.4.1.4), and so contain no internal buffering. (However, you can construct *processes* that perform buffering.)
* Channels support only read ("?") and write ("!") operations carrying data values. The operations behave in the same way as take and put.
* The most fundamental channels are one-to-one. They may be connected only to a single pair of processes, a writer and a reader. Multiple-reader and multiple-writer channels may also be defined.

### 4.5.1.2 Composition

Much of the elegance of CSP stems from its simple and analytically tractable composition rules. The "S" in CSP stands for *Sequential*, so basic processes perform serial computations on internal data (for example adding numbers, conditional tests, assignment, looping). Higher-level processes are built by composition; for a channel c, variable x, and processes P and Q:

|  |  |
| --- | --- |
| c?x -> P | Reading from c enables P |
| c!x -> P | Writing to c enables P |
| P ; Q | P followed by Q |
| P || Q | P and Q in parallel |
| P [ ] Q | P or Q (but not both) |

The choice operator P [ ] Q requires that P and Q both be communication-enabled processes (of formd?y -> R or d!y -> R). The choice of which process runs depends on which communication is ready: Nothing happens until one or both communications are ready. If one is (or becomes) ready, that branch is taken. If both are (or become) ready, either choice may be taken (nondeterministically).

### 4.5.1.3 JCSP

The JCSP package supports CSP-based design in a straightforward way. It consists of an execution framework that efficiently supports CSP constructs represented via interfaces, classes, and methods, including:

* Interfaces ChannelInput (supporting read), ChannelOutput (supporting write) andChannel (supporting both) operate on Object arguments, but special versions for intarguments are also provided. The principal implementation class is One2OneChannel that supports use only by a single reader and a single writer. But various multiway channels are also provided.
* Interface CSProcess describes processes supporting only method run. Implementation classes Parallel and Sequence (and others) have constructors that accept arrays of otherCSProcess objects and create composites.
* The choice operator [ ] is supported via the Alternative class. Its constructor accepts arrays with elements of type Guard. Alternative supports a select method that returns an index denoting which of them can (and then must) be chosen. A fairSelect method works in the same way but provides additional fairness guarantees — over the course of multiple selects, it will choose fairly among all ready alternatives rather than always selecting one of them. The only usages of Alternative demonstrated below use guard typeAltingChannelInput, which is implemented by One2OneChannel.
* Additional utilities include CSProcess implementations such as Timer (which does delayed writes and can also be used for time-outs in Alternative), Generate (which generates number sequences), Skip (which does nothing at all — one of the CSP primitives), and classes that permit interaction and display via AWT.

### 4.5.1.4 Dining philosophers

As a classic demonstration, consider the famous Dining Philosophers problem. A table holds five forks (arranged as pictured) and a bowl of spaghetti. It seats five philosophers, each of whom eat for a while, then think for a while, then eat, and so on. Each philosopher requires two forks — the ones on the left and right — to eat (no one knows why; it is just part of the story) but releases them when thinking.

[**Figure 4-43**](javascript:popUp('graphics/04fig43.gif'))

The main problem to be solved here is that, without some kind of coordination, the philosophers could starve when they pick up their left forks and then block forever trying to pick up their right forks which are being held by other philosophers.

There are many paths to a solution (and yet more paths to non-solution). We'll demonstrate one described by Hoare that adds a requirement (enforced by a Butler) that at any given time, at most four philosophers are allowed to be seated. This requirement suffices to ensure that at all times at least one philosopher can eat — if there are only four philosophers, at least one of them can get both forks. This solution does not by itself ensure that all five philosophers eventually eat. But this guarantee can be obtained via use of Alternative.fairSelect in the Butler class to ensure fair processing of seating messages.

We'll use a simple, pure CSP style where all channels are one-to-one and messages have no content (using null for messages). This puts a stronger focus on the synchronization and process construction issues. The system is composed of a College with five Philosophers, five Forks, and one Butler(standing in the bowl of spaghetti!), connected using One2OneChannels.

[**Figure 4-44**](javascript:popUp('graphics/04fig44.gif'))

Since everything must be either a process or a channel, forks must be processes. A Fork continuously loops waiting for a message from one of its users (either its left-hand or right-hand philosopher). When it gets a message from one indicating a fork pick-up, it waits for another indicating a fork put-down. (While it might be more tasteful to indicate pick-ups versus put-downs via different kinds of messages or message contents, this protocol using null messages suffices.)

In JCSP, this can be written as:

class Fork implements CSProcess {

private final AltingChannelInput[ ] fromPhil;

Fork(AltingChannelInput l, AltingChannelInput r) {

fromPhil = new AltingChannelInput[ ] { l, r };

}

public void run() {

Alternative alt = new Alternative(fromPhil);

for (;;) {

int i = alt.select(); // await message from either

fromPhil[i].read(); // pick up

fromPhil[i].read(); // put down

}

}

}

The Butler process makes sure that at most N-1 (i.e., four here) philosophers are seated at any given time. It does this by enabling both enter and exit messages if there are enough seats, but only exitmessages otherwise. Because Alternative operates on arrays of alternatives, this requires a bit of manipulation to set up. (Some other utilities in JCSP could be used to simplify this.) The exitchannels are placed before the enter channels in the chans array so that the proper channel will be read no matter which Alternative is used. The fairSelect is employed here to ensure that the same four philosophers are not always chosen if a fifth is also trying to enter.

class Butler implements CSProcess {

private final AltingChannelInput[ ] enters;

private final AltingChannelInput[ ] exits;

Butler(AltingChannelInput[ ] e, AltingChannelInput[ ] x) {

enters = e;

exits = x;

}

public void run() {

int seats = enters.length;

int nseated = 0;

// set up arrays for select

AltingChannelInput[ ] chans = new AltingChannelInput[2\*seats];

for (int i = 0; i < seats; ++i) {

chans[i] = exits[i];

chans[seats + i] = enters[i];

}

Alternative either = new Alternative(chans);

Alternative exit = new Alternative(exits);

for (;;) {

// if max number are seated, only allow exits

Alternative alt = (nseated < seats-1)? either : exit;

int i = alt.fairSelect();

chans[i].read();

// if i is in first half of array, it is an exit message

if (i < seats) --nseated; else ++nseated;

}

}

}

The Philosopher processes run forever in a loop, alternating between thinking and eating. Before eating, philosophers must first enter their seats, then get both forks. After eating, they do the opposite. The eat and think methods are just no-ops here, but could be fleshed out to (for example) help animate a demonstration version by reporting status to JCSP channels and processes that interface into AWT.

class Philosopher implements CSProcess {

private final ChannelOutput leftFork;

private final ChannelOutput rightFork;

private final ChannelOutput enter;

private final ChannelOutput exit;

Philosopher(ChannelOutput l, ChannelOutput r,

ChannelOutput e, ChannelOutput x) {

leftFork = l;

rightFork = r;

enter = e;

exit = x;

}

public void run() {

for (;;) {

think();

enter.write(null); // get seat

leftFork.write(null); // pick up left

rightFork.write(null); // pick up right

eat();

leftFork.write(null); // put down left

rightFork.write(null); // put down right

exit.write(null); // leave seat

}

}

private void eat() {}

private void think() {}

}

Finally, we can create a College class to represent the parallel composition of the Forks,Philosophers, and Butler. The channels are constructed using a JCSP convenience function createthat creates arrays of channels. The Parallel constructor accepts an array of CSProcess, which is first loaded with all of the participants.

class College implements CSProcess {

final static int N = 5;

private final CSProcess action;

College() {

One2OneChannel[ ] lefts = One2OneChannel.create(N);

One2OneChannel[ ] rights = One2OneChannel.create(N);

One2OneChannel[ ] enters = One2OneChannel.create(N);

One2OneChannel[ ] exits = One2OneChannel.create(N);

Butler butler = new Butler(enters, exits);

Philosopher[ ] phils = new Philosopher[N];

for (int i = 0; i < N; ++i)

phils[i] = new Philosopher(lefts[i], rights[i],

enters[i], exits[i]);

Fork[] forks = new Fork[N];

for (int i = 0; i < N; ++i)

forks[i] = new Fork(rights[(i + 1) % N], lefts[i]);

action = new Parallel(

new CSProcess[ ] {

butler,

new Parallel(phils),

new Parallel(forks)

});

}

public void run() { action.run(); }

public static void main(String[ ] args) {

new College().run();

}

}

## 4.5.2 Further Readings

CSP has proven to be a successful approach to the design and analysis of systems that can be usefully expressed as bounded sets of identityless, interfaceless processes communicating via synchronous channels. CSP was introduced in:

Hoare, C. A. R. *Communicating Sequential Processes*, Prentice Hall, 1985.

An updated account appears in:

Roscoe, A. William. *The Theory and Practice of Concurrency*, Prentice Hall, 1997.

Several of the texts listed in Chapter 1 (including the book by Burns and Welling in 1.2.5.4) discuss CSP in the course of describing constructs in occam and Ada.

Other related formalisms, design techniques, languages, and frameworks have adopted different base assumptions that adhere more closely to the characteristics of other concurrent systems and/or to different styles of analysis. These include Milner's CCS and pi-calculus, and Berry's Esterel. See:

Milner, Robin. *Communication and Concurrency*, Prentice Hall, 1989.

Berry, Gerard. "The Foundations of Esterel", in Gordon Plotkin, Colin Stirling, and Mads Tofte (eds.), *Proof, Language and Interaction*, MIT Press, 1998.

As package support becomes available for these and related approaches to concurrent system design, they become attractive alternatives to the direct use of thread-based constructs in the development of systems that are best viewed conceptually as collections of active objects. For example, *Triveni* is an approach based in part on Esterel, and is described in:

Colby, Christopher, Lalita Jategaonkar Jagadeesan, Radha Jagadeesan, Konstantin L ufer, and Carlos Puchol. "Objects and Concurrency in Triveni: A Telecommunication Case Study in Java", *USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.

Triveni is supported by a Java programming language package (see the online supplement). Among its main differences from CSP is that active objects in Triveni communicate by issuing events. Triveni also includes computation and composition rules surrounding the interruption and suspension of activities upon reception of events, which adds to expressiveness especially in real-time design contexts.

### Footnote

1. As of this writing, a similar class is scheduled to be supported in an upcoming SDK release.