CS697B-f07 Class Notes

Steve Revilak

September 2007 – December 2007

These are Steve Revilak's notes from cs697 Distributed and Concurrent Systems, taught by Jun Suzuki during Fall 2007.

Part 1

Processes

1.1 Lecture – 9/4/2007

1.1.1 General Course Notes

Although it's not officially a seminar course, this course will be presented in a similar fashion. There are no textbooks, and there will be no exams. We'll spend a portion of our time reading papers, and presenting the paper material to the class.

We will also do individual projects (later in the semester).

Grading will be done as follows:

- 40% programming assignments
- 40% course projects
- 20% presentations

Class discussions will cover programming concepts in Java. However, we are free to use any language we want for programming assignments.

Course web site: `http://www.cs.umb.edu/~jxs/courses/2007/697`

Email address for homework submission: `umasscs697b@gmail.com`

1.1.2 Processes in an Operating System

- A *process* is a container (execution environment) for a program
- All modern operating systems support multi-tasking
- At any moment a single CPU (actually a single core) can only execute a single program.
- Multitasking with one CPU is *pseudo-parallelism*. The operating system runs each process for a short period of time (a time slice), multiplexing across many processes. This multiplexing gives the illusion that processes are executing concurrently, even though it's not true parallelism.

  Process execution is not in a completely serial manner.

1.1.3 Process States: 5-state model

The simplest process model uses five states, as shown in Figure 1.1.

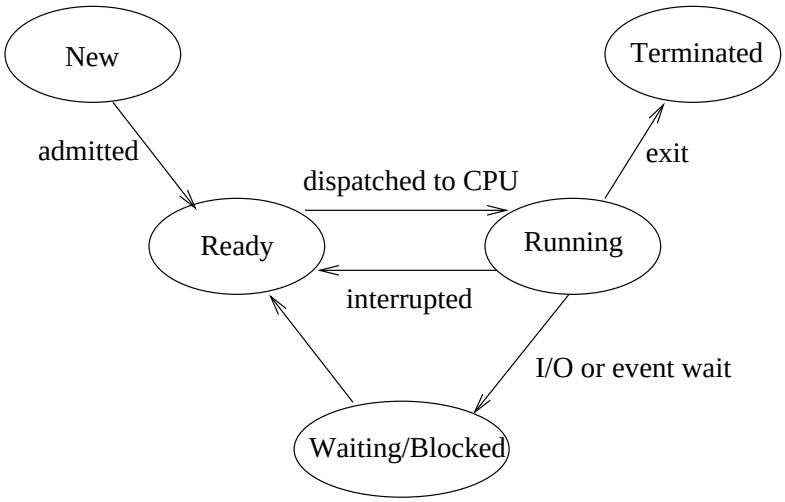

Figure 1.1: 5-state process model

- The ready → running transition occurs when the CPU becomes available.
- The running → waiting transition occurs when the process makes an I/O request. The process has to wait for a disk read, or perhaps for a network read.
- The waiting → ready transition occurs after the I/O request completes.

The key points here:

- The CPU is multiplexed.
- As a process runs, it changes state.

Example 1.1.3.1: Consider a simple unix pipeline:

```
$ cat file1 file2 file3 | grep UMB
```

If we dissect the state changes, we'll have something like this:

- cat enters the ready state
- cat transitions to running, then makes an I/O request
- cat enters the waiting state (for disk I/O)
- grep enters the ready state
- grep transitions to running, then to waiting (waiting to read cat's stdout)
- cat finishes reading the files, and moves to the running state
- cat runs, combining the files
- cat's stdout goes to grep's stdin
- grep moves from waiting (for stdin read) to ready, and then to running
- After cat writes its output, it goes to the terminated state
- grep does its work in the running state
- grep moves to the terminated state.

Example 1.1.3.2: Let's look at a simple Java I/O operation:

```
InputStreamReader fp = new InputStreamReader(System.in);
fp.read();
```

Here,

- The java process moves from new to ready to running
- An I/O request is made from the running state

- The I/O request causes java to enter the waiting state (waiting for stdin input)
- The user enters some text
- Java moves from waiting to ready to running.

1.1.4 To-Do

For next class, look over the `java.nio` apis. Maybe some of the stuff in `java.util.concurrent.*` as well.

1.2 Lecture – 9/6/2007

1.2.1 Architectural View of an OS

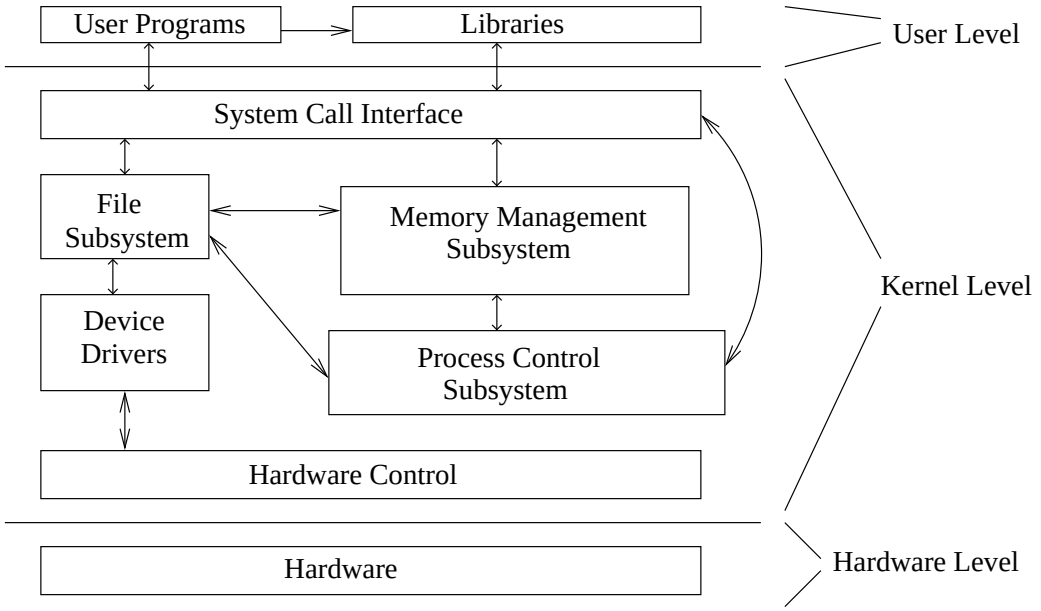

Figure 1.2: Architectural View of An Operating System

Figure 1.2.1 shows an architectural view of an operating system.

The *process management subsystem* can be further decomposed into subsystems for the following tasks:

- Inter-process Communication
- Process Control

The *system call interface*:

- Provides functions that allow user programs (and libraries) to interact with the kernel.
- Provides functions to create, use, and delete objects in the kernel. This includes functions such as `fork`, `mkdir`, `kill`, and `wait`.
  "Kernel Objects" include things like processes, files, and network connections.

1.2.2 Process State Transitions

Suppose we have three processes, A, B, and C. The operating system provides a program called the *scheduler* whose job is to schedule CPU time for the processes. The simplest scheduler follows a "fair" policy, in which the scheduler allocates equal CPU time to all processes.

Each processes consumes some amount of memory space. This includes the scheduler, which is just another software component.

In a simple model (using physical memory only), our scheduler and three processes might be laid out as shown in Figure 1.3.

In Figure 1.3, the letters x, a, b, and c denote the base address of each process.

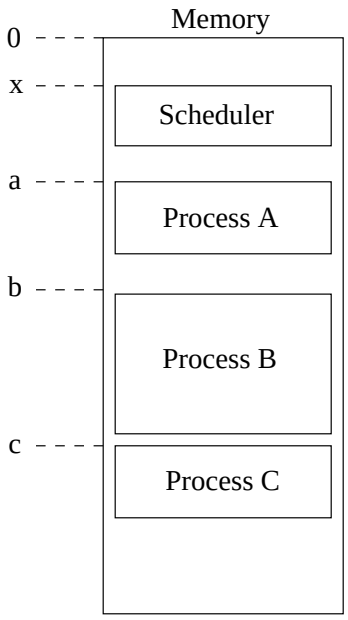

Figure 1.3: Memory Layout for Scheduler and Three Processes

Figure 1.4 shows a detailed view of the instruction space for process A. We consider a process to have N instructions in assembly language, where each instruction consumes one memory location (each instruction is allocated to a single address).

$$\begin{array}{|c|}\hline a+0 \\ a+1 \\ a+2 \\ \vdots \\ a+(N-1) \\ \hline\end{array}$$

Figure 1.4: Detailed View of Instruction Space for Process A

The OS has a register called a *program counter*, which points to the address of the instruction that's currently being executed. Running a process consists of executing a series of assembly language instructions.

If we think of the program counter as a base address + an offset, the interleaving of the scheduler + programs A, B, C will look something like figure 1.5.

In Figure 1.5, we're assuming a time slice of 5 instructions. You can see this by noting the timeout after tick 5, and again after tick 17. Process B lost it's turn on the CPU after tick 10, when it made an I/O request.

You can also see the scheduler running between each process (ticks 6–7, 11–12, and 18–19).

Figure 1.2.2 shows another view of the interleaving presented in Figure 1.5. In Figure 1.2.2 we see how the states of each process change as they are interleaved.

A few things to note about Figure 1.2.2:

- During ticks 6–7, 11–12, 18-19, and 25–26, the scheduler is running. During these time periods, processes A, B, C are ready or blocked (not running).
- For ticks 27–31, note how the scheduler skipped B's turn, and picked C to run instead. B was blocked for I/O, and therefore ineligible to run.

Cpu Tick	Process Instruction	
1	a+0	
2	a+1	
3	a+2	
4	a+3	
5	a+4	Time Out
6	x+0	
7	x+1	
8	b+0	
9	b+1	
10	b+2	I/O Request
11	x+0	
12	x+1	
13	c+0	
14	c+1	
15	c+2	
16	c+3	
17	c+4	Time Out
18	x+0	
19	x+1	

Figure 1.5: Interleaved Execution of Scheduler and Processes A, B, C

Time	Process A	Process B	Process C
1–5	**Running**	Ready	Ready
6–7	Ready	Ready	Ready
8–10	Ready	**Running**	Ready
11–12	Ready	Blocked	Ready
13–17	Ready	Blocked	**Running**
18–19	Ready	Blocked	Ready
20–24	**Running**	Blocked	Ready
25–26	Ready	Blocked	Ready
27–31	Ready	Blocked	**Running**

Figure 1.6: State Transition View of Process A, B, C

1.2.3 Implementing the Five-State Process Model

From the discussion so far, we see that our scheduler has to perform at least two types of book keeping:

- The scheduler must keep track of which processes are ready. Because we've been assuming a fair scheduler, the *ready queue* can be a simple FIFO structure.
- The scheduler must keep track of which processes are blocked waiting for an event to occur. We've been saying I/O event, but there are other types of event waits as well.

  The I/O queue does not have to be a first-in first-out structure, since I/O completions probably won't happen in the same order that I/O requests are made, particularly for requests made from network resources.

  Furthermore, when an I/O event occurs, the OS must easily be able to determine which process made the request, and which process to unblock.

Figure 1.7 depicts the data flow chores for the OS scheduler.

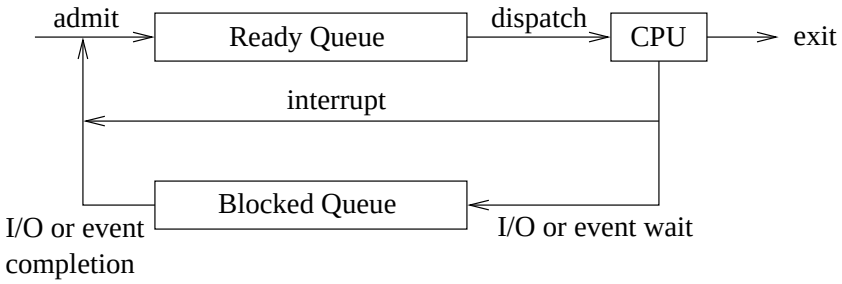

Figure 1.7: Data Flow Implementation for 5-state process layout

It's not uncommon for Operating Systems to use separate blocked queues for each type of event. This leads to a data flow like the one shown in Figure 1.8

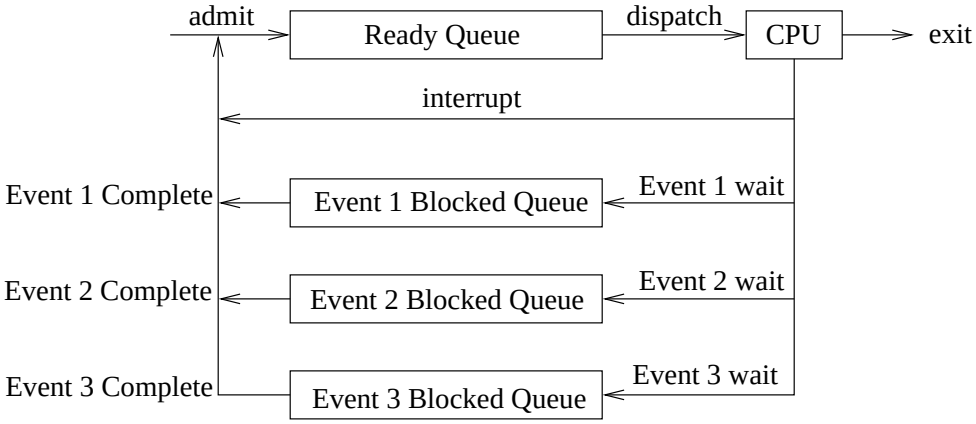

Figure 1.8: Five State Process Data Flow With Multiple Event Queues

1.2.4 Memory Layout and Swapping

So far, we've assumed that processes fit entirely within memory. But what if the size of the processes exceeds the physical memory size? Most operating systems handle this through the use of *virtual memory*, also known as *swapping*.

Virtual memory resides on disk. The OS 'swaps' a process out by moving some portion of the process space from main memory to disk.

Blocked processes are usually the ones that are swapped out first.

In order to support swapping, we'll need an additional process state (representing a process that is swapped out to disk). This gives us the state transitions shown in Figure 1.9.

In Figure 1.9, the "suspended" state represents a process that has been swapped out to disk.

The six-state model is too simplistic, though. When bringing a swapped process back into memory (activation), we'd prefer to bring in a process that isn't blocked in event wait. We'll handle this by adding one more state, giving the model shown in Figure 1.10. This model uses two states to represent swapped processes: processes that are swapped and waiting, and processes that are swapped and runnable.

The significant parts of the seven-state model are:

Ready The process is in main memory and available for execution

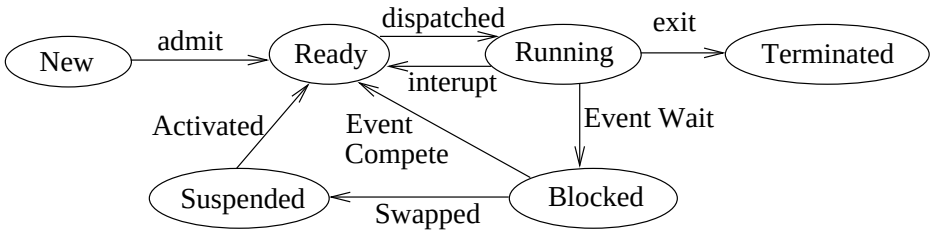

Figure 1.9: Six-State Process (1 Swap State)

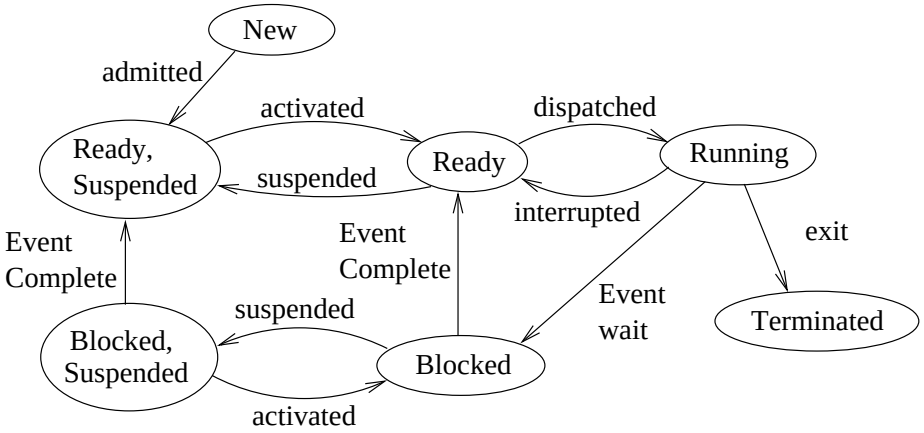

Figure 1.10: Seven-State Process (2 Swap States)

Blocked The process is in main memory and blocked on an event.

Blocked, Suspended The process is in virtual memory and blocked on an event.

Ready, Suspended The process is in virtual memory, but available for execution as soon as it is loaded into main memory.

Again, the scheduler has to know which swapped processes are waiting for events, and which ones are not.

1.2.5 First Homework Assignment

Class notes contain four homework problems. Mail in answers by Thursday, 9/13/2007. PDF or Word Documents are fine.

1.3 Process Notes from Stallings

These are a few notes taken from Chapter 3 of *Operating Systems* 4th edition, William Stallings, Prentice Hall, 2001.

1.3.1 Process Queues

Pages 111-112 suggest two structures for process queues:

1. A queue where each entry is a pointer to a process data structure.
2. A linked list, where each linked-list node is a process structure.

For the most part, Stallings tends to assume the latter.

In the five-state process model, there are two queues of interest: the Ready Queue and the Blocked Queue.

Ready Queue In lieu of a priority scheme, a simple FIFO queue is perfectly adequate. (pg. 117)

Waiting Queue There's more to the waiting queue than a simple FIFO. When an event completes, one has to locate the process associated with the event – the the order of event completion isn't going to follow the order in which the processes became blocked.

1.3.2 Process States

New State

In the new state, the control structures needed to manage the process have been created in memory. However, the process itself has not been loaded into memory.

Exit State

In the exit state, the process is no longer running, but the data structures used to manage it still reside in memory. At this point, one can (for example) extract accounting information for the process.

Suspended State(s)

A suspended process has been swapped out, making more physical memory available for other processes.

As noted earlier, more than one swapped state is necessary. In particular, we'd like to distinguish between:

- Processes that are in the suspended state, but still waiting for an event. If activated, such a process cannot be moved to the ready state.
- Processes that have been suspended, but are not waiting for an event. When activated, these can be moved directly to the ready state.

Thus, we have two independent concepts:

1. Whether the process is waiting for an event (blocked or ready)
2. Whether the processes has been swapped out of main memory (suspended or not suspended).

These concepts are represented by four states (given below).

Ready State

The process is in main memory, and available for execution.

Blocked State

The process is in main memory, and waiting for an event.

Blocked/Suspended

The process is swapped out, and waiting for an event.

Ready/Suspended

The process is in secondary memory (swapped), but will be ready to run as soon as it is brought back into main memory.

1.3.3 Process State Transitions

Running → Ready

Most commonly, a process makes the Running → Ready transition when it has used its CPU time allotment.

Other reasons for making this transition: The process is preempted by another process with higher priority; or, the process may voluntarily relinquish control of the CPU.

Running → Blocked

A process makes the Running → blocked state when it requests something for which it must wait. For example, an I/O request for a local file, or from an network connection.

This transition may also occur when a process is waiting to receive an event from another process.

Blocked → Ready

The blocked → ready transition occurs when a process is waiting for an event, and the event occurs. The process is ready to run, as soon as the CPU becomes available.

Blocked → Blocked/Suspend

A blocked process is swapped out, to make more room for processes that are not blocked.

Blocked/Suspend → Ready Suspend

Occurs when a blocked (and suspended) process was waiting for an event, and the event occurs.

Ready/Suspend → Ready

This transition occurs when the OS moves a process from swap to main memory. The process can run as soon as the CPU becomes available.

Ready → Ready/Suspend

Normally, an operating system prefers to swap out a blocked process. However, sometimes the only way the OS can free up a sufficiently large chunk of memory is to swap a ready process. In this case, a process takes the ready → ready/suspend transition.

This transition can also occur when the operating system suspends a lower priority process because the OS feels that a higher priority (but blocked) process will be runnable soon.

1.4 Lecture – 9/13/2007

New address for project submissions: `umasscds@gmail.com`.

Start thinking about what you'd like to do for a project. We'll have a lot of latitude – for example, if there's a piece of middleware that we'd like to try working with, that would be an acceptable project. It's also okay to choose a project that would tie in with other course work (or phd projects).

1.4.1 Processes and Resources

What information does an operating system need in order to manage processes. Different operating systems use different structures, but the bulk of it comes down to four things: memory, I/O, files and processes.

Figure 1.11 shows a conceptual layout of these four tables. We've left Memory, I/O, and Files as opaque structures. The process table is shown with more detail. The process table has one entry per process. Process table slots contain pointer to process images (the actual process memory space), in addition to other information about the process.

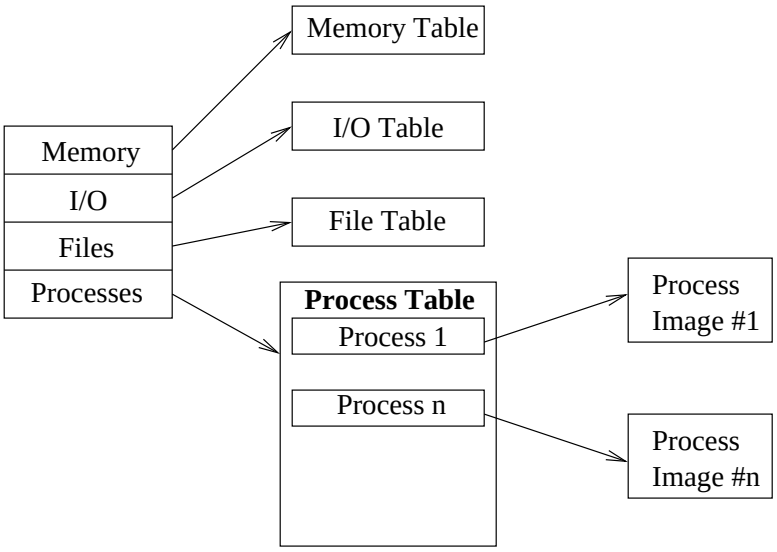

Figure 1.11: Conceptual Layout of Four Main OS Tables

1.4.2 Core Dumps

Core dumps are one mechanism for examining the image of a process. They're often generated as a result of system crashes. By examining a core dump with a debugger, we can determine what it was doing when it crashed.

1.4.3 Operating System Tables

Referring back to Figure 1.11, we see four primary tables used by the operating system. Although we've drawn them as distinct structures, these tables are often cross references (for example, the process table will often contain references to the memory and I/O tables, corresponding to memory and I/O devices used by the process).

These tables are usually created during the boot-up sequence. A partial list of things the OS needs to set up these tables includes:

- Memory size
- Number of CPUs
- What types of I/O devices are available (and how many)
- User-provided attributes about the system hardware (e.g. BIOS).

We'll concentrate on the process table, giving only brief summaries of the other three.

1.4.4 Memory Table

Memory tables keep track of both main memory and virtual memory.

Memory tables keep track of what regions are free, and which have been assigned to processes (and to the specific processes assigned to each region).

The memory table also records any protection attributes. For example, memory can be marked as read-only or shared.

1.4.5 I/O Table

The I/O table keeps track of what I/O devices are present on the system hardware. It also tracks I/O device use, and the status of I/O operations.

The I/O table also manages the memory space used as the source and destination of I/O operations.

For example, the I/O table keeps track of `stdin`, `stdout` and `stderr`.

1.4.6 File Tables

The file table keeps track of files (for example, which files are opened).

1.4.7 Process Table

The process table contains links to individual process images (among other things).

Each process has a *process image*, which consists of four parts:

User Program The binary image of the program being executed by the user.

User Data Mutable user memory. This includes both the stack and dynamically allocated memory.

Kernel Stack The kernel stack is used to keep track of system calls and the parameters passed to them. It behaves like the user stack – the difference is that it keeps track of user calls only.

Process Control Block (PCB) Data needed by the operating system to control a process (i.e. the process attributes).

Figure 1.12 shows a typical process control block layout.

The process control block is one of the most important operating system structures. Among the things it contains are:

- The process identifier (pid). Most operating systems assign a unique numerical identifier to each process.

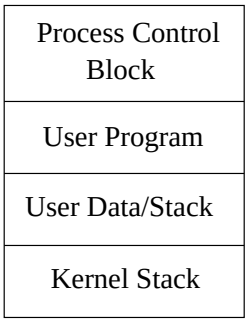

Figure 1.12: Process Control Block Layout

- CPU state information for the process. This includes program registers, and the contents of the program counter. (The program pointer holds the location of the instruction being executed).
- Process control information. This includes a list of any events that the process is waiting for, scheduling information, a list of open files, and the status of any file operations.

Each process images exists in main memory or in virtual memory. The memory region occupied by a process image may be contiguous or non-contiguous.

Memory regions are typically assigned in *pages*. Pages may be allocated in fixed or variable sizes.

In order to run, the entire process image must be loaded into main memory.

When swapping, the OS might not move the entire process image out to disk. Instead, it might move only enough to free up memory for the task at hand. By only swapping out part of the process, the OS reduces the amount of time needed to swap out an reload the process.

Figure 1.13 shows a revised process table diagram. Here, we see three queues, and several processes in different states. The processes are chained together by linked lists (we could also use arrays).

Although Figure 1.13 shows only three process states, a real operating system would contain others.

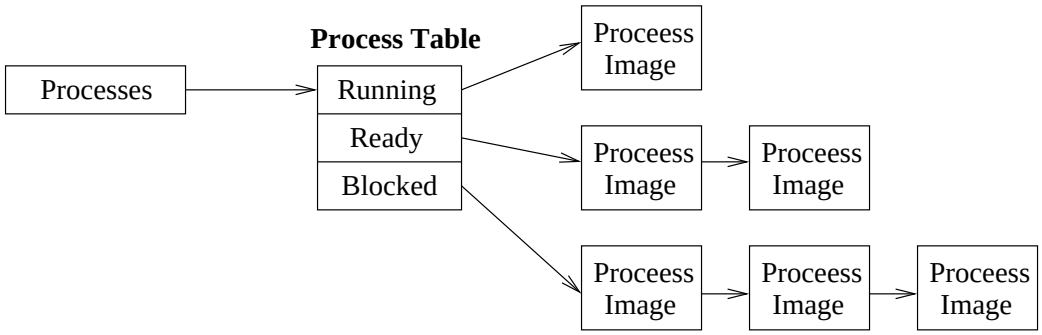

Figure 1.13: Revised Process Table Diagram

1.4.8 Processes and CPU Switches

Given what we've discussed so far, we can see that the operating system needs to perform a specific set of actions when processes are moved into and out of the running state. These operations will occur during *process preemption* or during a *context switch*.

These tasks include saving and loading the process image (including the such things as the contents of CPU registers, the program counter, etc).

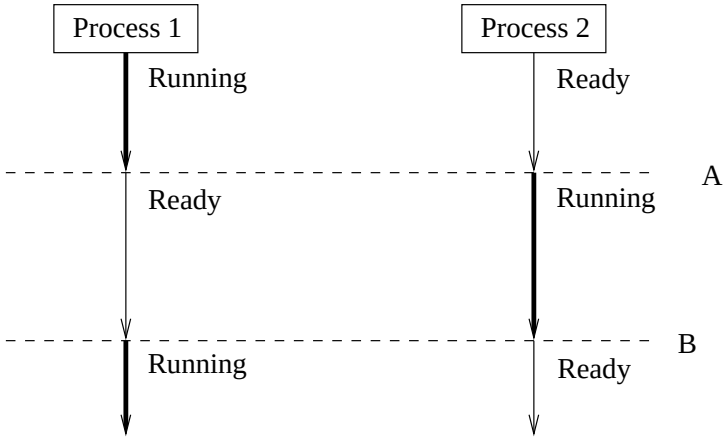

Figure 1.14: Diagram of a Context Switch

Figure 1.14 shows a diagram of a process context switch. At the point in time labeled A, the operating system must save the image of process one and load the image of process two. The opposite happens at time B: the operating system saves the image of process two, and loads the image of process one.

1.4.9 Process Creation

Processes begin life in the New State. While in the new state, the operating system will

- Assign a new (unique) process id to the process.
- Add the process's entry to the process table. Recall that the process table contains one entry per process.
- Allocate memory space for the process.
- Initialize all elements of the process images.
- Place the process in the appropriate queue (typically the end of the Ready/Suspended Queue).

1.5 Lecture – 9/18/2007

1.5.1 Process (Context) Switches

There are 3 primary circumstances that cause process switches (also known as *context switches*) to occur.

1. **System Calls**. The program makes a system call (`read()`, etc). This causes control to shift from user level to kernel level. While the system call is executing, the program will generally be in a blocked state.
2. **Interrupts**. Interrupts occur independently of user programs. A typical example of an interrupt is I/O completion – an I/O device becomes ready, and an interrupt handler is invoked to fetch the available data. If a process is waiting for this I/O event, then the process can go from blocked → Ready when the interrupt handler completes.

   Clock interrupts are another common interrupt type.
3. **Traps**. Traps represent errors or exceptional conditions. (For example, illegal file access).

   If a trap is fatal, the process will move into the terminated state, and exit shortly thereafter.

   If the trap is not fatal, then an error code will typically be returned to the program.

1.5.2 Interrupts

Nearly all operating systems have a notion of *interrupt cycle*. At regular intervals of time, the OS polls to see if any interrupts have occurred, calling the appropriate handlers for each interrupt.

If there is a pending (i.e. - unhandled) interrupt, then the OS saves the image of the currently running process and invokes the appropriate interrupt handler.

The interrupt handler acknowledges and services the interrupt.

1.5.3 Interrupt Handlers

Interrupt handlers are (usually) very short and simple pieces of code.

As part of servicing, the interrupt handler will also acknowledge the interrupt (so that the OS knows its been handled), and the interrupt handler will also report any error conditions.

Clock interrupts are slightly special – these cause CPU control to be given to the scheduler.

Recall that the operating system polls for interrupts at fixed intervals (the interrupt cycle). What if an interrupt handler can't finish before the cycle is over? In other words, what do we do if interrupt #2 occurs while the handler for interrupt #1 is still running?

In general, operating systems must be able to deal with multiple interrupts. This is typically done with a priority scheme – a high priority interrupt (like the clock) can preempt a lower-priority interrupt.

1.5.4 The Speed of Context Switches

In our last lecture, we looked briefly at the actions taken when a context switch occurs (see Figure 1.14 on page 17).

Context switching is "pure overhead" – during this time, no user programs are running. Therefore, we'd prefer context switches to happen as quickly as possible.

Factors that affect the speed of context switches:

- The hardware and Operating system.
- The speed of memory (how fast is memory access)
- CPU speed
- The number of CPU registers
- The system bus speed.
- Disk access speed (if the context switch involves moving things into or out of virtual memory).

1.5.5 Process Scheduling

The act of determining which process to run next is called *process scheduling*.

There are three main types of process scheduling: (1) short term, (2) medium-term, and (3) long-term.

A single operating system can employ all three types of scheduling. Different types of scheduling are appropriate for different process state transitions. In the seven-state model, we'd have:

- **Short Term**. Appropriate for Ready ↔ Running.
- **Medium Term**. Ready/Suspended ↔ Ready, and Blocked/Suspended ↔ Blocked.
- **Long Term**. New → Ready/Suspended, and New → Ready.

Medium-Term Scheduling

Medium-Term scheduling usually involves deciding which process to move into or out of virtual memory. In some operating systems, medium-term scheduling is entirely implemented by the virtual memory system.

Long-Term Scheduling

Long-term scheduling determines which process to *admit*. (ie - to move from the New → Ready, or New → Ready/Suspended).

If a process has just been created, that doesn't necessarily mean it will be admitted right away. For example, there might be too many processes running right now, whereby the new one will have to wait.

Factors considered by a long-term scheduler:

- How many processes there are. If we have more processes, each process will receive a smaller (relative) period of time to run. The more processes we have, the greater the competition for CPU time.
- Does the percentage of idle CPU time exceed a certain threshold?
- Are any processes terminating?

Short-Term Scheduling

Short-Term scheduling decides which ready process should run next. Short-Term scheduling is the most prevalent, and it's the kind we'll focus on the most.

Short term schedulers make very frequent, and very fine-grained decisions.

There are two types of short-term schedulers: preemptive and non-preemptive.

Preemptive Short-Term Schedulers Processes are allowed to run until (1) the use up their time slice, and are moved into the ready state or (2) an event causes the process to wait (I/O request, interrupt, system call, etc).

Non-preemptive Short-Term Schedulers Processes are allowed to run until (1) they finish, (2) they yield control of the CPU, or (3) they make an I/O request. A scheduler will not kick a process off the CPU because it's been running too long.

Non-preemptive schedulers are often found in real-time systems. Real-time systems have to be very deterministic, and schedulers are often non-deterministic (with respect to execution time). Also, in a real-time system, the overhead imposed by a preemptive scheduler may simply be too great.

1.5.6 Evaluation Criteria for Schedulers

- CPU utilization. (How well does the scheduler cause the CPU to be utilized?). More precisely, it's process execution time divided by total CPU time.
- Throughput. How much work is done per unit of time?
- Turnaround Time. How long is the period between process submission and process termination.
- Waiting time. How long is the time between process submission and when the process first runs on the CPU?
- Response Time. How long is the between process submission and when the process generates its first output?

1.5.7 7-State Queueing Diagram

Figure 1.15 shows a queuing diagram for the 7-state process model. This is the answer to problem HW4.

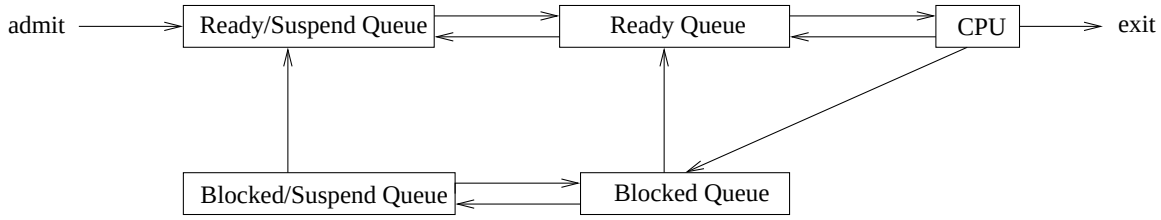

Figure 1.15: Queueing Diagram for 7-State Process Model

1.6 Stallings - Ch 9, Uniprocessor Scheduling – 9/20/2007

The long-term scheduler makes its decisions when a new process is added.

The medium-term scheduler is part of the swapping function.

The short-term scheduler actually decides what process to run next (i.e. - which process will have the next turn on the CPU).

1.6.1 Long-Term Scheduler Decisions

The primary decisions made by the long-term scheduler are:

- Can the operating system handle another running process
- Which new process will be admitted to run?

The long-term scheduler may limit the multiprogramming level of a system for the sake of providing better service guarantees to the currently running set of processes.

1.6.2 The Short-Term Scheduler

- The short term scheduler is also known as a *dispatcher*.
- The short-term scheduler is invoked any time there is an opportunity to preempt the currently running process. This opportunity can come in the form of a clock interrupt, an I/O interrupt, a system call, or a signal.

1.6.3 Scheduler Evaluation

There are two broad criteria for evaluating schedulers: user oriented and system oriented. User-oriented aspects focus on the user's perception of the system (responsiveness, etc.). System-oriented aspects focus on effective and efficient use of resources (determinism, predictability, etc).

Schedulers can also be evaluated quantitatively (based on performance characteristics), or qualitatively (user perception).

The design of a scheduler always involves compromise. One cannot optimize every dimension simultaneously.

The *selection function* is a common component of any short-term scheduler. This function decides which process to run next.

1.6.4 Scheduling Algorithms

This section briefly summarizes several of the scheduling algorithms explained in Stallings.

First Come First Serve (FCFS)

FCFS is the simplest policy. Processes are run in the order that they appear in the ready queue. First Come First Serve tends to favor CPU bound processes over I/O bound processes.

FCFS is a non-preemptive scheduling algorithm.

Round Robin

Round robin schedulers are built on the notion of *time-slicing*. The scheduler preempts processes at regular intervals, based upon the system clock.

The *time quantum* is the principal design choice for a round robin scheduler.

Round robin schedulers are most effective in general-purpose time sharing systems.

Round Robin schedulers tend to favor CPU bound processes over I/O bound processes. A CPU-bound processes will usually have the opportunity to use its full time quantum. I/O bound processes are likely to be moved off the CPU (blocked) before their quantums complete.

Shortest Process Next (SPN)

SPN is a non-preemptive scheduling algorithm. The scheduler selects the process that is expected to have the shortest run time. SPN tends to increase responsiveness, but tends to decrease predictability.

An SPN scheduler must have some kind of estimate of how long a process will take.

Shortest Running Time (SRT)

SRT is a preemptive version of shortest process next. Again, the scheduler needs some kind of estimate of how long a given process will run.

Highest Response Ratio Next (HRRN)

HRRN is based on the formula

$$R = \frac{w + s}{s} \tag{1.1}$$

where

R	is the response ratio
w	is the time spent waiting for the CPU, and
s	is the expected service time

An HRRN scheduler simply chooses the process with the highest value of R.

Feedback

Feedback is an adaptive approach that doesn't require knowing how long a process will run. (Shortest process next, Shortest running time, and Highest response ratio next all require some notion of process length).

Feedback penalizes those processes which have been running the longest.

Processes are given the highest priority level when they begin. As the process executes, its priority is gradually reduced. Once a process has been assigned the lowest priority level, it stays there.

For long-lived processes, turnaround time can increase a lot, since the scheduler is always favoring younger processes.

Fair Share Scheduling

Fair share takes a different approach to process scheduling. Rather than looking at processes as individuals, fair share looks at a group of processes as a single set. The scheduler makes its decisions based on the set as a whole.

The original idea was to group processes according to the user that invoked them – each user got their "fair share" of system resources.

This concept can be extended to groups of users.

Several unix systems use variations on this idea.

1.7 Lecture – 9/20/2007

1.7.1 Scheduler Metrics

Recall a few of the scheduler metrics we discussed in the last lecture

$$\text{wait-time} = tstart - tadmit \tag{1.2}$$

$$\text{turnaround-time} = tfinish - tadmit \tag{1.3}$$

$$\text{throughput} = nprocs/time \tag{1.4}$$

Where

$tstart$	clock tick where process begins executing
$tadmit$	clock tick where process is admitted
$tfinish$	clock tick where process finishes (last tick of execution)

1.7.2 First-Come First-Serve Scheduling

When processes become ready, they enter a FIFO ready queue. The scheduler always chooses the process at the head of the queue to run next. Once a process gets a turn on the CPU, it runs without preemption.

Under First-Come First-Serve, processes will execute sequentially.

Consider the set of processes shown in table 1.1.

Process	Arrival Time	Duration
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Table 1.1: Set of Five Processes

Under FCFS, the scheduling of these processes will be as shown in Figure 1.16.

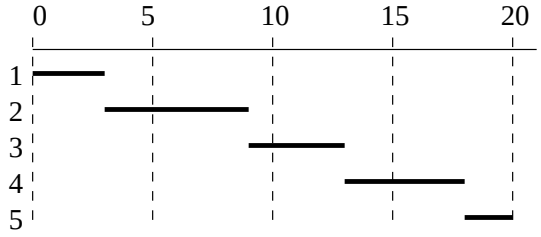

Figure 1.16: FCFS Scheduling of Processes in 1.1

Under FCFS we have

$$\text{avg waiting time} = \frac{0+1+5+7+10}{5} = 4.6 \text{ ms}$$

$$\text{avg turnaround time} = \frac{3+7+9+12+12}{5} = 8.6 \text{ ms}$$

$$\text{throughput} = 5/20 = 0.25 \text{ procs/ms}$$

The order and length of process execution can have a dramatic effect of these metrics.

Consider the following set of three processes:

Process	Arrival Time	Duration	tstart
1	0	24	0
2	1	3	24
3	2	3	27

For these three processes, FCFS gives us

$$\begin{aligned}\text{avg waiting time} &= \frac{0+23+25}{3} &&= 16 \text{ ms}\\ \text{avg turnaround time} &= \frac{24+26+28}{3} &&= 26 \text{ ms}\\ \text{throughput} &= 3/30 &&= 0.1 \text{ procs/ms}\end{aligned}$$

Now, Suppose the three processes arrive in a different order:

Process	Arrival Time	Duration	tstart
2	1	3	0
3	2	3	3
1	0	24	6

With this ordering, FCFS gives

$$\begin{aligned}\text{avg waiting time} &= \frac{0+2+4}{3} &&= 2 \text{ ms}\\ \text{avg turnaround time} &= \frac{3+5+28}{3} &&= 12 \text{ ms}\\ \text{throughput} &= 3/30 &&= 0.1 \text{ procs/ms}\end{aligned}$$

This illustrates the following: depending on how long a process consumes the CPU, FCFS can exhibit a marked lack of fairness. Short processes can't run while a long process has control of the CPU.

These examples also illustrate the high variability of waiting and turnaround time.

First come first serves does not treat long and short-lived processes fairly.

1.7.3 Round Robin

Round Robin scheduling uses clock-based preemption (i.e. - time slicing). This algorithm reduces the penalty for short-lived process that we see under FCFS. Round robin tends to be fairer than FCFS.

With round robin, the key design issue is the length of the time slice. If time slices are too short, then we'll incur too much overhead interrupting processes and performing context switches. On the other hand, if the time slice is too long, then round robin degenerates into FCFS.

Effects of Time Slice on Round Robin

Let's look at a few example. Consider our group of five processes with a time slice of 1ms.

Process	Arrival Time	Duration
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

Round robin with a 1ms time slice handles this as follows

Time	On CPU	Queue	remarks
0	1	-	
1	1	-	
2	2	1	proc 2 admit
3	1	2	proc 1 exit
4	2	3	proc 3 admit
5	3	2	
6	2	4, 3	proc 4 admit
7	4	3, 2	
8	3	2, 5, 4	proc 5 admit
9	2	5, 4, 3	
10	5	4, 3, 2	
11	4	3, 2, 5	
12	3	2, 5, 4	
13	2	5, 4, 3	
14	5	4, 3, 2	proc 5 exit
15	4	3, 2	
16	3	2, 4	proc 3 exit
17	2	4	proc 2 exit
18	4	-	
19	4	-	proc 4 exit

(note: this chart assumes that process admission happens before the running processes is moved back into the ready queue

If you count the number of changes in the second column, you'll see that there were seventeen context switches.

Now, let's work out the same example with a 4ms time slice.

Time	On CPU	Queue	remarks
0	1	-	proc 1 admit
1	1	-	
2	1	2	proc 2 admit, proc 1 exit
3	2	-	
4	2	3	proc 3 admit
5	2	3	
6	2	3, 4	proc 4 admit
7	3	4, 2	
8	3	4, 2, 5	proc 5 admit
9	3	4, 2, 5	
10	3	4, 2, 5	proc 3 exit
11	4	2, 5	
12	4	2, 5	
13	4	2, 5	
14	4	2, 5	
15	2	5, 4	
16	2	5, 4	proc 2 exit
17	5	4	
18	5	4	proc 5 exit
19	4	-	proc 4 exit

With a 4ms time slice, we see that there are only 6 context switches. This would lead us to conclude that a 1ms time slice is too small.

What would happen if we used a 6ms time slice? Our longest-lived process runs for 6ms; with a 6ms time slice, round robin would degenerate into FCFS.

Rules of Thumb for Time Quantums

For a round robin scheduler, we can use a few rules of thumb when selecting a time slice:

- The time quantum should be slightly longer than the typical process execution. (Otherwise, we do too much context switching).
- The time quantum should be shorter than the longest process executions. (Otherwise, we degenerate into FCFS).
- Typical values are 10–100 ms

The 10–100ms range comes from trial and error, based on what operating systems designers have done over time.

Another Fairness Issue

We've seen that for short vs long processes, Round Robin behaves more fairly than First Come First Serve. But, Round Robin has a different fairness issue: round robin does not treat I/O bound and CPU bound processes fairly.

With round robin, CPU-bound processes will generally have the opportunity to use their full time quantum. By contrast, I/O bound processes will usually become blocked before their quantum is up. Thus, I/O bound processes frequently execute for *less* than their full quantum.

The net effect is that CPU bound processes monopolize the CPU, while I/O bound processes tend to suffer.

For our next homework assignment, we'll read about an algorithm that proposes a solution to this dilemma.

1.7.4 Normalized Turnaround Time

In our FCFS section, we've seen how ordering and process length can have a big effect on turnaround time. There is a similar metric called *derived turnaround time* that's less affected by this.

For a single process, normalized turnaround time (NT) is defined as

$$NT = \frac{\text{turnaround time}}{\text{execution time}} \tag{1.5}$$

1.8 A Few References for HW6

A few of the references I found for HW6:

- List of PC interrupts.
  `http://www.pcguide.com/ref/mbsys/res/irq/num.htm`
- Good article on writing device drivers. Use This!
  `http://developer.apple.com/hardwaredrivers/customusbdrivers.html`
- Also, *I/O Kit Fundamentals* has a really good overview of interrupt event types.
- Getting started on Hardware drivers
  `http://developer.apple.com/referencelibrary/GettingStarted/GS_HardwareDrivers/index.html`
- Hardware (MPIC) controller. Probably too little detail to be useful.
  `http://developer.apple.com/documentation/Hardware/DeviceManagers/pci_srvcs/pci_cards_drivers/PCI_BOOK.141.html#10323`
- BSDCon paper on Darwin Kernel. Talks about Interrupts in a SMP setting.
  `http://www.usenix.org/events/bsdcon02/full_papers/gerbarg/gerbarg_html/index.html`

Also: article on using MPI with Darwin:
`http://developer.apple.com/hardwaredrivers/hpc/mpionmacosx.htm`

1.9 Notes in *Fairness in Processor Scheduling* – 9/23/2007

Haldar, S. and Subramanian, D. K. 1991. Fairness in processor scheduling in time sharing systems. *SIGOPS Oper. Syst. Rev.* 25, 1 (Jan. 1991), 4-18. DOI `http://doi.acm.org/10.1145/122140.122141`

These are notes taken from

```
@article{122141,
 author = {S. Haldar and D. K. Subramanian},
 title = {Fairness in processor scheduling in time sharing systems},
 journal = {SIGOPS Oper. Syst. Rev.},
 volume = {25},
 number = {1},
 year = {1991},
 issn = {0163-5980},
 pages = {4--18},
 doi = {http://doi.acm.org/10.1145/122140.122141},
 publisher = {ACM Press},
 address = {New York, NY, USA},
}
```

1.9.1 Overview

The primary contribution of this paper: presenting an algorithm that addresses the lack of fairness between CPU-bound and I/O-bound processes, as exhibited by Round Robin scheduler algorithms.

Scheduling can be thought of as a variation on the mutual exclusion problem (in this case, the resource is the CPU). A good scheduler should exhibit fairness.

Definition 1.9.1.1 (*k*-fair): A scheduler is *k-fair* if, once process p requests a resource r, and another process q which has requested r later than p, could not use r more than k times ahead of p.

(I understand what the author's are trying to say, but it seems like the last p should be a q ....)

1.9.2 1-fair Schedulers

1-fair Schedulers are a proposed solution to the 1-fair problem. 1-fair schedulers work as follows:

- Requests for resource r are partitioned into 2 groups: the *current* group and the *waiting* group.
- New Requests go into the waiting group.
- Requests in the current group are serviced randomly
- When no unserviced requests remain in the current group, then we swap groups. The waiting group becomes the current group, and we have a new (empty) waiting group.

This solution is guaranteed to be 1-fair, even if process selection within the current group is done unfairly.

1.9.3 Conventional Round Robin (CRR) Schedulers

A *conventional round robin* (CRR) scheduler is fair, provided that all processes are of the same type: all CPU-bound or all I/O bound.

We call a process *CPU-bound* if it mainly performs computational work, and occasionally uses I/O devices.

We call a process *I/O-bound* if it spends more time using I/O devices than it spends using the CPU.

Generally, I/O bound processes will have shorter *CPU bursts* than

1.9.4 Virtual Round Robin (VRR) Schedulers

The VRR scheduler tries to achieve a greater level of fairness between I/O and CPU-bound processes. VRR is based on the concept of *dynamically shrinking time quanta*. VRR retains much of the advantages of CRR.

VRR splits the ready queue into two new queues.

Main Queue Ready processes are placed on the main queue when (1) the process is created or (2) when the process is preempted after using its full time quantum.

Auxiliary Queue The auxiliary queue holds ready process that issued I/O commands and became blocked before their time quantum was up.

Suppose our time quantum is Q. A process in the main queue will be able to stay on the CPU for Q time, provided that it issues no blocking operations.

A process in the Aux. Queue will be able to stay on the CPU for $(Q - Q')$ duration, where Q' represents the time spent on the CPU before the process became blocked. For example, suppose our time Quantum was 10 ms, and a process p issued an I/O command after spending 5ms on the CPU. When p's I/O requests completes, p will be placed on the aux queue, and when p is chosen to run, p will be given $(10 - 5) = 5$ ms of CPU time.

The quantity $(Q - Q')$ is referred to as the *remaining time slice*.

With VRR, the scheduler employs the following algorithm when selecting a process to run next:

Algorithm 1 VRR Scheduling Algorithm

```
1: if aux queue is not empty then
2:     remove and schedule process at the head of the aux. Queue
3: else if Main queue is not empty then
4:     remove and schedule process at the head of the main queue
5: else
6:     idle
7: end if
```

1.10 Lecture – 9/25/2007

Last class, we covered two scheduling algorithms: First-come first-serve (FCFS), and round robin (RR). We saw that FCFS has a fairness issue with respect to long and short-lived processes. We saw that RR has a fairness issue with respect to CPU and I/O-bound processes.

In our homework assignment, we'll read about *virtual round robin*, and discuss how the algorithm addresses the fairness issues in RR.

1.10.1 Multi-Level Feedback Queue Scheduling (MFQ)

Multi-level feedback queue scheduling favors sort-lived processes over long-lived ones. It's usually implemented in a preemptive manner. Figure 1.17 shows the basic idea.

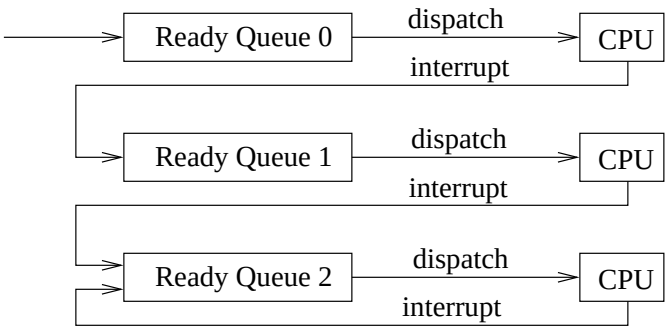

Figure 1.17: Queueing Diagram for Multi-Level Feedback Queueing

The different queues effectively implement different priority levels. As Figure 1.17 shows, the longer the process lives, the more its priority decreases. Eventually, the process reaches the lowest priority level and stays there.

The dispatcher for Multi-Level Feedback queue takes processes from Ready Queue 0. If no processes are in ready queue 0, then the scheduler takes a process from Ready Queue 1. If no process is in ready queue 1, then the scheduler takes a process from Ready Queue 2.

Our example uses three queues, but a real implementation could have more (or less).

Suppose we have three processes:

Process	Arrival Time	Execution time
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

The queueing of these processes will be handled as follows, assuming a time quantum of 1 ms.

Time	Running	RQ 0	RQ 1	RQ 2	Remarks
0	1				1 admit
1	1				
2	2			1	2 admit
3	2			1	
4	3			1, 2	3 admit
5	3			1, 2	
6	4			1, 2, 3	4 admit
7	4			1, 2, 3	
8	5			1, 2, 3, 4	5 admit
9	5			1, 2, 3, 4	5 exit
10	1			2, 3, 4	1 exit
11	2			3, 4	
12	3			4, 2	
13	4			2, 3	
14	2			3, 4	
15	3			4, 2	3 exit
16	4			2	
17	2			4	
18	4			2	4 exit
19	2				2 exit

There's more detail than this table reveals. For example, during the first few clock ticks:

- At the beginning of $t = 0$, Process 1 enters RQ0, and is dispatched to the CPU.
- At the end of $t = 0$, process 1 is interrupted, and moved to RQ1.
- At the beginning of $t = 1$, the scheduler finds no process in RQ0, so it looks at RQ1. Process 1 is at the head of RQ1, so it is dispatched.
- At the end of $t = 1$, process 1 is interrupted and moved into RQ2. Also, Process 2 is admitted, and added to RQ0. The scheduler prefers RQ0, so process 2 is dispatched to the CPU.

And so fourth.

MFQ tends to leave I/O bound processes in higher-priority queues. CPU bound processes will find their way into the lowest-priority queues faster.

MFQ has a possible starvation issue with long-lived processes. If lots a short processes are admitted very quickly, long-lived processes could wait for a long time before running.

1.10.2 MFQ-i

MFQ-i is a variation of MFQ that assigns different quantums to different queues. If our quantum was 2^i (where i represents the queue number), we'd have the following queue/time-quantum association:

Ready Queue	quantum
0	1 ms
1	2 ms
2	4 ms

Lets work out our earlier example with MFQ-i.

Time	Running	RQ 0 ($tq = 1$)	RQ 1 ($tq = 2$)	RQ 2 ($tq = 4$)	Remarks
0	1				1 admit
1	1				
2	1	2			2 admit, 1 exit
3	2		2		
4	3		2		3 admit
5	2		3		
6	2	4	3		4 admit
7	4		3	2	
8	5		3, 4	2	5 admit
9	3		4, 5	2	
10	3		4, 5	2	
11	4		5	2, 3	
12	4		5	2, 3	
13	5			2, 3, 4	5 exit
14	2			3, 4	
15	2			3, 4	
16	2			3, 4	2 exit
17	3			4	3 exit
18	4				
19	4				4 exit

Let's looks at the first few clock ticks in detail:

- $t = 0$. Process 1 enters RQ0, and is dispatched to the CPU for 1 ms. Once process 1 is interrupted, it's moved to ready RQ1.
- $t = 1$. No process is in RQ0, so the scheduler selects process 1 at the head of RQ1 to run for 2 ms.
- $t = 2$. Process 2 is admitted. Process 1 has not used its time quantum yet, so process 2 must wait. Process 2 is placed in RQ0.
- $t = 3$. Process 1 has exited, so process 2 is removed from RQ0 and dispatched.

1.10.3 Comparing RR and MFQ

If we compare RR and MFQ (both with fixed time quantums of 1ms),

- The number of context switches is similar
- MFQ has a smaller normalized turnaround time. This indicates a greater level of fairness for long and short-lived processes.

For MFQ-i ($tq = 2^i$), the normalized turnaround time indicates a better balance between long and short lived processes.

1.10.4 Variants of MFQ

There are many possible variations to MFQ.

One variation is to use FCFS instead of RR at the lowest-priority queue. In figure 1.17, this would amount to having the RQ2→ CPU transition be FCFS, while the others remain RR.

Another variation is to promote a processes priority based on some set of criteria. For example, if a process had stayed in RQ2 for too long, we might move it up to RQ1. Strategies like this help prevent starvation of long-lived processes.

1.10.5 Priority Scheduling

Several of the scheduling algorithms we've looked at have the notion of *priority*. Priority is determined by a set of *internal characteristics*. By "internal", we mean "something inside the operating system". Internal characteristics are measurable and quantifiable. For example, the length of time the process has been running, and whether it is I/O or CPU bound.

Internal characteristics are measured by running the process.

There are also *external characteristics*. External characteristics depend on the operating system. For example, system processes, vs. interactive processes, vs. batch processes, vs. real-time processes.

External characteristics are dictated by the operating system designer.

Some choices for the operating system designer:

- Is our scheduler preemptive or not?
- Do we use different ready queues? Do we assign different processes to different ready queues?
- Is the scheduling algorithm based on fixed or dynamic priorities?

Some issues for the operating system designer:

- How do we prevent starvation for lower-priority processes?
- Do we use a "promotion by aging" strategy?

1.10.6 Traditional Unix Schedulers

When we say "traditional unix scheduler", the term *traditional* refers to 4.3 BSD or SVR 3. Traditional schedulers have no concept of real-time processes. They distinguish only between time-sharing (interactive) and batch processes.

When we say *modern* unix scheduler, we're referring to 4.4 BSD or SVR4. These operating systems recognized real-time, and other types of processes.

Traditional schedulers:

- use MFQ with preemptive context switching, and round-robin at the lowest priority queue.
- assign the same time quantum to all priority queues.
- use dynamic priority assignment.

1.10.7 Unix Priorities

Unix uses a priority scheme with values from 0–127. Lower numbers indicate higher priorities.

- priorities 0–49 are reserved for kernel processes (or processes running in kernel mode)
- priorities 50–127 are designated for user processes

At a *conceptual* level, 128 priorities means 128 process queues. However, an operating system may not implement them that way.

When user code executed a system call, the priority will increase to kernel level. When the system call completes, priority will return to the previous user-level priority.

1.10.8 Scheduling & Process Control Blocks

In a previous lecture, we mentioned that a process control blocks contained (a) process identification information, (b) CPU state information, and (c) process scheduling information. Priority is a component of process scheduling information.

The PCB contains the following scheduling information:

1. The current scheduling priority (`pri`),
2. The user mode priority (`usrpri`),
3. A Measure of recent CPU utilization (`cpu`), and
4. The process's nice value (`nice`)

`usrpri` is derived from `cpu` and `nice`.

Current Scheduling priority

The current scheduling priority (`pri`) is used to determine which process to dispatch to the cpu.

In a user program, `pri` can change, depending on whether the user program is executing a system call. Specifically

- A user process normally executes at priority `usrpri`. So, `pri` = `usrpri`.
- During a system call, the priority will be elevated. This causes `pri` to decrease, but `usrpri` remains the same.
- When the system call returns, the priority goes back to normal: `pri` = `usrpri`.

For example, on BSD systems, all I/O operations occur at priority 20.

User mode Priority

User mode priority (`usrpri`) dictates the running priority of user-level code. `usrpri` is derived from CPU utilization (`cpu`), and the process nice value (`nice`). To put it more quaintly

$$usrpri = f(cpu, nice)$$

CPU Utilization

The operating system keeps a record of how much CPU time a process has used. Usually, this number starts at zero when the process is created, and increases with every clock interrupt.

10 ms is a ballpark figure for the frequency of clock interrupts.

Nice value

`nice` gives users a way to affect the priority of their processes. A normal user can request a lower priority for their processes, but only a superuser can increase the priority.

Nice values range from $0 \leq$ `nice` ≤ 30. The default value is 20.

Higher nice values mean lower priority.

Every second, user priority is updated as follows:

$$cpu_j(i) = DR \times cpu_j(i-1) \tag{1.6}$$

$$usrpri_j(i) = PUSER + \frac{cpu_j(i)}{2} + nice_j \tag{1.7}$$

where

i	represents a time interval (clock interrupt tick)
j	represents a process
DR	is a *decay rate*. SVR 3 uses $DR = 0.5$.
$NPROC$	is the baseline priority for user processes. A constant; something like 50.

1.11 Lecture – 9/27/2007

1.11.1 Traditional Unix Scheduler Example

Recall our "traditional unix" scheduler formulas:

$$cpu_j(i) = \frac{cpu_j(i-1)}{2}$$
$$usrpri_j(i) = 50 + \frac{cpu_j(i)}{2} + nice_j$$

Let's work out an example with three processes, where

$DR = 0.5$	decay rate
$nice = 0$	
$PUSER = 50$	(base priority for user processes)
$tq = 1$ sec	time quantum

Although our example uses three processes, we're only going to show one of them. This is a round-robin scheduler, so processes two and three will look similar, just shifted later in time by 1 second for process two, and two seconds for process three.

For process one, `usrpri` and `cpu` values are shown in Figure 1.18.

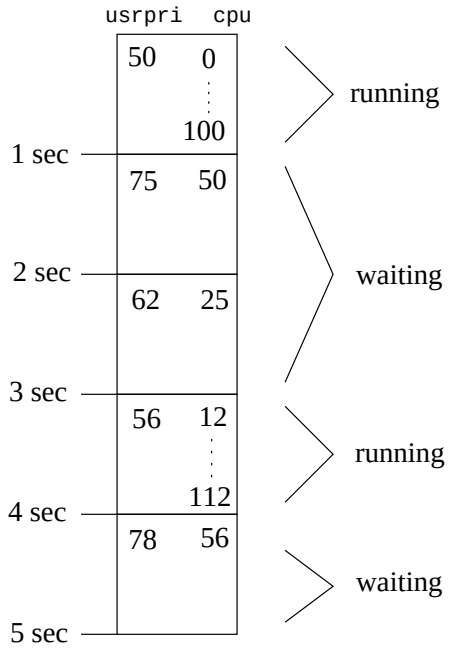

Figure 1.18: Traditional Unix Scheduler Example

Recall that our time quantum is 1 second. CPU values are updated with every clock interrupt. Clock interrupts occur every 10 ms, so we can see the `cpu` value increasing by 100 whenever the process is in the running state.

Note: if this process were to block for I/O, it would have a "higher" priority when it was unblocked and runnable. Because the process became blocked, it ran for less time, and the `cpu` value decreased less. The higher CPU value translates into a higher priority.

1.11.2 The 4.3 BSD Scheduler

The 4.3 BSD scheduler has a priority range of 0–127, but 32 ready queues. This means that each ready queue contains a range of priorities. For example, the highest priority queue will hold processes with priorities 0–3. Although there are 32 *physical* queues, there are still 128 *conceptual* queues.

Within each ready queue (or *run queue*, in BSD terminology), round robin (with preemption) is used.

4.3 BSD uses a time quantum of 100 ms.

Within a single queue, the process with the highest priority will be scheduled first, even if it's not at the head of the queue. I suppose we can think of it this way: when queued, a process leaps over any processes in the same queue which have a lower priority. So, this isn't a 'pure' round-robin.

The scheduler users a processes `pri` field when deciding which process to schedule next.

4.3 BSD uses MFQ across ready queues. If a running process has a higher priority than any queued process, then the running process will be allowed to keep running – even into the next time quantum. (Eventually, the `cpu` increase reduces the process's priority, and some other process will get a chance to run).

Process priority values are updated every second (like the example given earlier in this section).

Processes are promoted and demoted across run queues, based on their `pri` values. If a process p has priority u, then p will be placed in the ready queue whose priority range includes u.

BSD performs context switches in the following circumstances:

1. When a process blocks or exits
2. As part of round-robin based preemption.
3. As part of priority based preemption. Priority values are updated every second.
4. At I/O (event) completion.

1.11.3 Process Groups

Traditional unix schedulers assign priority values to each proceess, and each process is treated independently for scheduling purposes.

However, we can develop scheduling algorithms that treat groups of processes as a single unit. We call these *process groups*.

A process group may consist of:

- processes created by one particular application,
- processes created by one particular user,
- processes in the same session

The commands `ps -j` and `ps -Aj` include process groups in their output.

1.11.4 Fair Share Scheduling (FSS)

Fair Share scheduling (FSS) is a scheduling algorithm that considers process groups. FSS assigns the same (or similar) priorities to all processes in a group. For example, given four (equal) process groups, each group would receive 25% of the CPU time

Priority (`pri`) can vary among processes within a single process group.

FSS is an extension of the traditional unix scheduler.

FSS is based on the following set of equations:

$$cpu_j(i) = DR_p \times cpu_j(i-1) \quad \text{process cpu value} \tag{1.8}$$

$$gcpu_k(i) = DR_g \times gcpu_k(i-1) \quad \text{group cpu value} \tag{1.9}$$

$$userpri_j(i) = PUSER + \frac{cpu_j(i)}{2} + \frac{gcpu_k(i)}{4 \times W_k} \tag{1.10}$$

Above

i	denotes a time quantum
j	denotes a process
k	denotes a process group
DR_p	denotes the process decay rate
DR_g	denotes the group decay rate
W_k	is the weight assigned to group k

For weighting, we have $0 < W_k \leq 1$ and $\sum_k W_k = 1$.

If a group contains only one process, then we'll have $cpu_j(i) = gcpu_k(i)$.

In FSS, *cpu* and *gcpu* are incremented with each clock interrupt. *cpu* is incremented on the running process only. *gcpu* is incremented on *all* processes in the group, even if they're not currently running. So, if a process is sitting in a ready queue, *cpu* will not be incremented, but *gcpu* will be.

Part 2

Threads

2.1 Lecture – 10/2/2007

2.2 Processes vs. Threads

Processes are containers for execution of a program. A process provides an execution environment for a program.

The term *threads* is short for "threads of execution" or "threads of control". A thread is a finer-grained container than a process. Traditional processes have one thread of execution; modern processes can have many different threads of execution.

To review, the process image of a traditional process contains the process control block, the user program, user data, a user stack, and a kernel stack. The process image of a threaded processes will have similar information – but we'll add a few things that are specific to each thread.

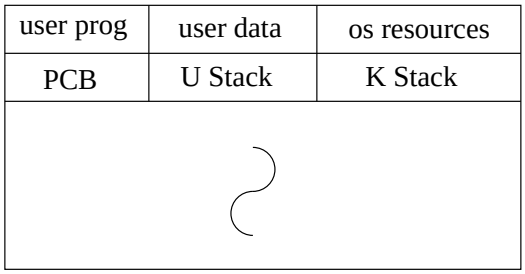

Figure 2.1: Traditional Process with a Single Thread

Figure 2.1 shows a diagram of a traditional process image, where the process executes as a single thread. In this diagram "U Stack" stands for user stack, and "K Stack" stands for kernel stack. "OS Resources" are things like file descriptors, etc.

Figure 2.2 shows a process image with three threads. Note that there is a set of information that is common to all threads in the process, and a set of information that is unique to each thread.

"U, K Stack" is an abbreviation for "User and Kernel Stack". Each thread has its own user and kernel stack.

"TCB" stands for *thread control block*.

In this model, the CPU is scheduled against threads. Threads allow concurrency within a single process.

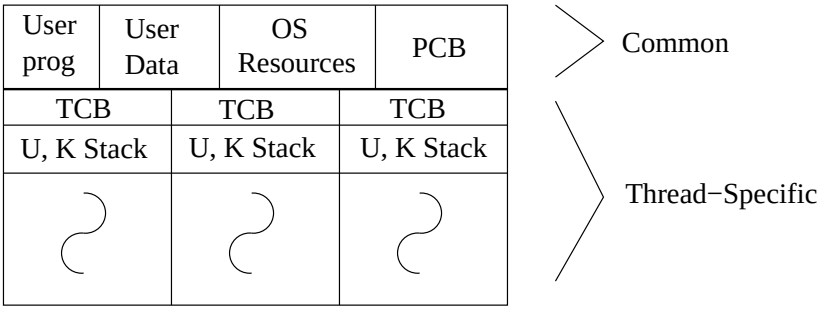

Figure 2.2: Process with Three Threads

In order to support this model, the operating system must have the concept of a *thread scheduler* as well as the concept of a *process scheduler*.

On a single CPU, threads give pseudo-parallelism. They're not executed in a truly concurrent fashion – they're executed round-robin.

2.2.1 Thread Control Blocks (TCB)

A thread control block contains:

- The thread identifier
- CPU registers
- The program counter (next instruction that the thread will execute).
- The event that the thread is waiting for (e.g. - an I/O event)
- Scheduling information

Within a single process, multiple threads will share:

- The user program code
- User data
- Operating system resources

2.2.2 Why Use Threads?

Threads give you a more granular level of concurrency control than processes do. Threads allow a program to do several things at once. For example, a web browser might respond to keystrokes, download files, render pages, and check for a newer version all at the same time.

The use of multiple processes was common 10–15 years ago, before threading technology was well-understood and well-developed.

Process creation is more heavyweight than thread creation. A generally excepted comparison:

- Creating a process is thirty times more expensive as creating a thread.
- Performing a process switch is five times more expensive than performing a thread context switch.

As illustrated in Figure 2.2, a process carries around a lot more data than a thread does.

In summary: why threads?

Responsiveness Threads allow a program to continue, even if part of the program (one thread) is blocked.

Conservation of Resources Threads share memory and resources with their parent process.

Efficiency Thread creation is less expensive than process creation. Thread context switches are less expensive than process switches.

2.2.3 Implementation Strategies for Threads

There are two main implementation strategies for threads:

1. Kernel-level threads (KTs)
2. User-level threads (UTs)

2.2.4 User-Level Threads

User-level threads are implemented by a library that runs in user space. The kernel is not aware of threading — it only understands processes. The user library contains the thread scheduler. This is illustrated by Figure 2.3.

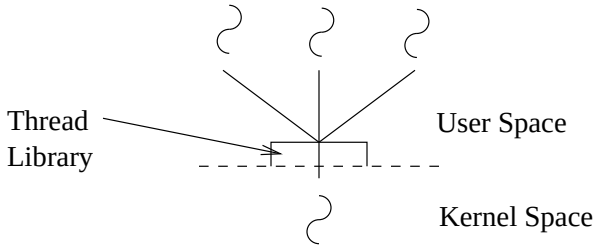

Figure 2.3: User-Level Threads

Pros of user threads:

- There is no overhead for mode changes (kernel mode vs user mode).
- The thread library can be made very portable.
- The thread scheduler can be application-specific.

The main disadvantage of user level threads: when one thread blocks, it blocks the whole process (because the OS sees it as a single process).

Examples of user-level threads:

- pthreads (POSIX Threads)
- Solaris "green" threads.

2.2.5 Kernel-Level Threads

In a kernel-level thread implementation, the kernel is aware of both threads and processes. If one thread blocks, other threads in the same process can continue to run. This can improve concurrency in a single program.

There are several implementation strategies for kernel-level threads. The most common is *one-to-one*: each user-level threads has a counterpart kernel-level thread. This is illustrated by Figure 2.4.

Linux, Windows, and Solaris Native Thread are examples of kernel-level thread implementations. (So, Solaris actually offers a hybrid thread environment).

Another technique for implementing kernel-level threads is *many-to-many*. This is also known as *thread multiplexing*. The number of kernel threads may be different than the number of user threads. But in general, the number of kernel threads will be smaller than the number of user threads.

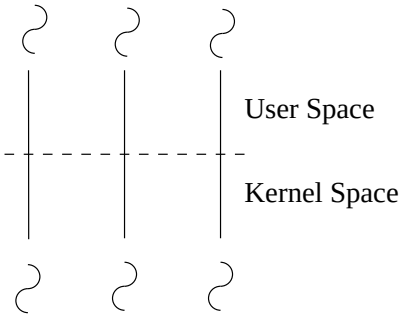

Figure 2.4: Kernel-Level Threads

Why many-to-many? To conserve system resources, the operating system may wish to put a limit on the number of kernel threads that can be created.

2.2.6 Java Threads

Any java program contains at least one thread (the one that executes `main()`). This thread (and several others) are implicitly created when the JVM starts.

There are four steps to create a Java thread:

- Define a class that implements the `Runnable` interface.
- Write a `run()` method for your class.
- Create a thread from the runnable object.
- Call the thread's `start()` method.

When you call `start()`, Java will call the `run()` method on your behalf. The OS-level thread is created when you call `Thread.start()`, not when you say `new Thread()`.

The JVM makes no guarantees about thread execution order, or about thread timing. As a programmer, you should assume that thread execution happens in a somewhat random order.

On my Mac, it looks like a simple "Hello World" program creates nine threads.

```
"Low Memory Detector" daemon prio=5 tid=0x0050a5a0 nid=0x1818c00 runnable
[0x00000000..0x00000000]

"CompilerThread0" daemon prio=9 tid=0x00509bb0 nid=0x1818800 waiting on condition
[0x00000000..0xf0b06450]

"Signal Dispatcher" daemon prio=9 tid=0x005097e0 nid=0x1818400 waiting on condition
[0x00000000..0x00000000]

"Finalizer" daemon prio=8 tid=0x00508fd0 nid=0x180fc00 in Object.wait()
[0xf0a04000..0xf0a04b30]
        at java.lang.Object.wait(Native Method)
        - waiting on <0x265802b0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:116)
        - locked <0x265802b0> (a java.lang.ref.ReferenceQueue$Lock)
        at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:132)
        at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:159)

"Reference Handler" daemon prio=10 tid=0x00508be0 nid=0x180ec00 in Object.wait()
[0xf0983000..0xf0983b30]
```

```
        at java.lang.Object.wait(Native Method)
        - waiting on <0x26580990> (a java.lang.ref.Reference$Lock)
        at java.lang.Object.wait(Object.java:474)
        at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)
        - locked <0x26580990> (a java.lang.ref.Reference$Lock)

"main" prio=5 tid=0x00501390 nid=0x1804600 waiting on condition [0xf07ff000..0xf07fff30]
        at java.lang.Thread.sleep(Native Method)
        at Bar.main(Bar.java:4)

"VM Thread" prio=9 tid=0x00508400 nid=0x1803c00 runnable

"VM Periodic Task Thread" prio=9 tid=0x0050bc50 nid=0x1819000 waiting on condition

"Exception Catcher Thread" prio=10 tid=0x005015a0 nid=0x1804a00 runnable
```

2.3 MPI

Trying out this implementation: `http://www.open-mpi.org/software/ompi/v1.2/downloads/openmpi-1.2.4.tar.gz`

Some documentation: `http://www.mpi-forum.org/docs/mpi2-report.pdf`

Nice Tutorial: `http://www.llnl.gov/computing/tutorials/mpi/`

Tips for debugging: `http://www.csafe.utah.edu/Information/Documentation/LongAnswers/gdb.html`

2.4 Lecture – 10/4/2007

2.4.1 Java Thread States

The java language specification defines six thread states:

- `NEW`
- `BLOCKED`
- `RUNNABLE`
- `WAITING`
- `TIMED_WAITING`
- `TERMINATED`

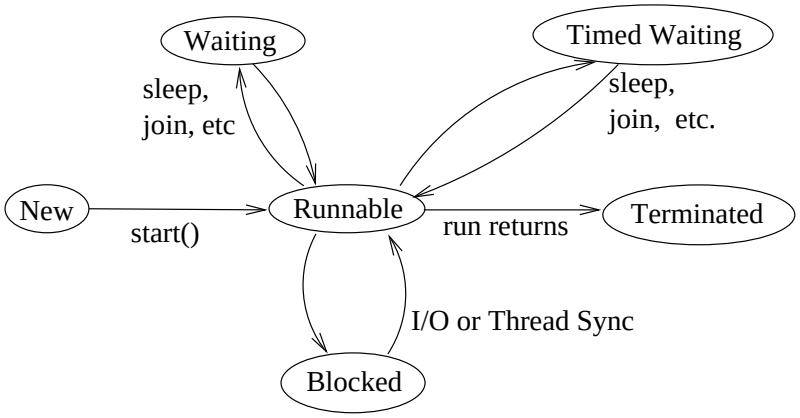

Figure 2.5: Java Thread State Transition Diagram

Java's `RUNNABLE` state does not distinguish between "ready" and "running". A runnable thread might be on the CPU, or it might be ready, and waiting to run on the CPU.

Transitions to and from the WAITING (and TIMED_WAITING) states are caused by the calls `notify`, `wait`, `sleep`, `join`, etc. If your thread specifies a time argument to these functions (for example `t.join(1000)`), then your thread will go into TIMED_WAITING. Otherwise, your thread will go into the WAITING state.

The enumeration class `Thread.State` lists all of the threads states.

If `t` is a thread,

- `t.getState()` gives the current Thread.State.
- `t.isAlive()` states whether the thread is alive.

`isAlive()` returns true if the thread is in the RUNNABLE, BLOCKED, WAITING, or TIMED_WAITING states. `isAlive()` returns false if the thread is in the NEW or TERMINATED states.

Threads are in the NEW state before `start()` has been called.

Threads are in the TERMINATED state after `run()` returns.

2.4.2 Thread Termination

Java threads may be terminated in one of two way: (1) implicit termination and (2) explicit termination.

Implicit termination occurs when the threads `run()` method returns normally.

Explicit termination occurs when an outside class "advises" the thread to exit (and the thread complies with this request). The most common techniques for explicit termination are (1) flags and (2) the `interrupt()` method.

Example of using a flag for explicit termination:

```
private boolean done = false;
public void setDone() {
  done = true;
}

public void run() {
   while (! done) {
      // ...
   }
}
```

As you can see, the `done` flag is strictly advisory. Once you set it, you have to wait until the thread notices that it's been set.

Interruption is another way to explicitly terminate a thread. Like the flag method, interruption is advisory.

```
public void run() {
  while (! isInterrupted()) {
    try {
      // ...
      Thread.sleep(1000);
    }
    catch (InterruptedException e) {
       break;
    }
  }
}
```

In this example interruption will disrupt the threads `sleep()`. Because some timed wait methods handle interruption by raising an `InterruptedException`, interruption can be faster than a flag. If using a flag, one has to wait until the thread is done sleeping.

Note that even if `interrupt()` is called when this thread is not sleeping, the while condition test will still notice the interrupt.

Note that `isInterrupted()` clears interrupt status.

In Java, explicit thread termination is *always* advisory. There is no way to forcibly kill a thread.

2.4.3 Stacks and Frames

Each java thread has its own stack (visible only to that thread).

Thread stacks work in the usual sense. Stack frames are pushed when methods are called, and popped when methods return. The stack contains one frame per method call.

Stack frames contain local variables, the thread's program counter, and compiler temporaries to hold the result of intermediate computations. For example, given `x = f() * g()`, we need temporaries to hold the result of the intermediate function calls; these temporaries are used to compute `x`.

The thread class contains several methods for examining stack frames. For example: `dumpStack()`, `getStackTrace()`, and `getAllStackTraces()`.

2.4.4 Homework Assignments

HW9 (Due 10/11/2007).

Java gives us two ways to create threads: implement `Runnable`, or extend `Thread`. Functionally, these two methods are the same. However, it is more common to use the former method. Why? (Hint: why does java have interfaces).

HW10 (Due 10/11/2007).

Rewrite `InterruptedTask3.java`. Instead of using `dumpStack()`, use `getStackTrace()` to print out the frame information.

HW11 (Due 10/18/2007).

Write a threaded GUI stop watch. The program should take a number of seconds as a command-line argument. This number will act as the initial value of a countdown timer.

The program should draw a window as shown in Figure 2.6.

Figure 2.6: GUI Layout for Timer Window

Use a separate thread to count the timer down. If the user presses the stop button, terminate the count-down thread using one of the techniques discussed today. If the user starts the timer (after stopping it), create a new thread to continue the countdown.

The timer should stop at zero, and reset itself.

HW12 (Due 10/18/2007).

Write a threaded program to compute Fibonacci numbers. Given n, the program should compute and display the first n number of the Fibonacci sequence. For example, `Fib(5)` should output `0, 1, 1, 2, 3, 5`, Note that there are six numbers (we're calling the first number '0').

This program should use two threads. One for main, and one to compute the Fibonacci sequence. Main should use `join()` to wait for the sequence computation to finish.

Main should create a collection and pass the collection to the thread that computes the Fibonacci sequence. The sequence thread stores data in the collection, and main reads the data out of the collection when the computation is done. In other words, the collection that holds the results of the computation must be *shared* between the two threads.

The Fibonacci computation must be done using recursion.

2.5 Lecture – 10/9/2007

2.5.1 Race Conditions

A *race condition* occurs when multiple threads read and write shared data at the same time. This can mess up the internal state of objects pretty badly. There is the potential for race conditions anytime we have unprotected access to shared data.

In order to prevent race conditions, we need to synchronize access to shared data.

In a race condition:

- There are no guarantees for the order of thread execution.
- There are no guarantees for the results of shared data.

2.5.2 Thread Synchronization

In order to prevent race conditions, we must synchronize data access among threads.

Definition 2.5.2.1 (Atomic Code): *Atomic code* is a set of statements whose execution cannot be interrupted. Within a block of atomic code, intermediate results are visible only to the thread executing the atomic code – these intermediate results are not visible to any other thread.

In other words, it's a critical section.

For example:

```
/* pseudo code for an atomic block */
atomic {
  a = 0;
  a = a + 3;
}
```

Threads that are not in the atomic code will never see the value `a = 0`.

If thread A is executing atomic code, then all other threads must wait until A is finished before they can execute the same set of statements.

Locks are the traditional mechanism for guarding atomic code.

Here's an example of java locking

```
aLock = new ReentrantLock();
aLock.lock();
 // atomic code goes here
aLock.unlock();
```

See the `java.util.concurrent.locks` package for more information on this class.

Important: aLock must be a unique object. Don't do something stupid, like declaring it as local variable.

2.5.3 Locks

Once a thread t calls `Lock.lock()`, then t "owns" the lock. Until t calls `Lock.unlock()` on the same lock object, no other thread can lock that object. If the locks are guarding a critical section, then the critical section is safe from race conditions.

Threads that are waiting to acquire a lock are in the `BLOCKED` state.

The most common locking idiom is this:

```
class MyClass {
  Lock lock = new ReentrantLock();
  // ...
  public void f() {
      lock.lock();
      try {
        /* critical section here */
      }
      finally {
        lock.unlock();
      }
  }
}
```

Putting `unlock()` in a finally block is *very* important. Suppose unlock wasn't in a finally block – if the critical section code threw an exception, then the lock would never be released; it would stay locked forever.

2.5.4 Thread Schedulers

There are two common methods for handling lock waits in the Java scheduler. The particulars vary with the JVM implementation.

1. The blocked thread is periodically woken up by the scheduler. It tries to run, but if the lock is not available, the thread goes back to the blocked state. This is a polling approach.
2. The blocked thread remains blocked until the lock is released. When the lock is released, the JVM notifies the blocked thread, waking it up to run. This is the event notification approach.

2.5.5 Nested Locking

Nested locking occurs when a thread holds a lock l, and then reacquires l through (a) a recursive call to the same method, or (b) a call to another method which uses the same lock.

An example of (a):

```
public int f(int x) {
  lock.lock();
  try {
     if (x == 0) {
        return 0;
     }
     else {
        f(x - 1);
     }
  }
  finally {
    lock.unlock();
  }
}
```

An example of (b):

```
Lock aLock = new ReentrantLock();
public void f() {
  aLock.lock();
  try {
    // ...
    g();
  }
  finally {
    aLock.unlock();
  }
}

public void g() {
  aLock.lock();
  try {
    // ...
  }
  finally {
    aLock.unlock();
  }
}
```

If thread t holds re-entrant lock l, then t is allowed to "lock" l again (but no other thread can lock l while t has it locked).

In order for this sort of thing to work properly, the number of `lock` calls *must* match the number of `unlock` calls.

2.5.6 For Next Class

Next class, we'll talk about deadlocks.

Read over javadoc for Barriers, Semaphores, and Read/Write locks.

2.6 Lecture – 10/11/2007

2.6.1 Deadlock

The general scenario for *deadlock*: two threads are waiting for each other, but at the same time, they are also preventing each other from continuing.

Condition Objects are one technique for avoiding deadlock. Suppose thread t_1 holds a lock, but needs to wait for thread t_2 to perform a task. t_1 temporarily releases its lock, and uses a condition object to wait for t_2. When t_2's task is performed, t_2 notifies t_1 via the condition object. t_1 reacquires its lock and proceeds.

The simplest form of this uses methods of `Object`.

```
/* thread #1 */
synchronized(this) {
  wait();
  // do stuff
}

/* thread #2 */
synchronized(this) {
   // ...
   notify();
}
```

Above, thread 1 acquires the monitor on the `this` object, then calls `wait`. Wait implicitly releases the monitor. Thread 2's `notify` call ends thread 1's wait. `wait` will not return until (1) notify() has been called and (2) Thread 1 has reacquired the monitor object on `this`.

The key to understanding this scenario is knowing how `wait` works.

- Wait can only be called when a thread owns an objects monitor.
- Wait implicitly releases the monitor.
- The waiting thread must be unblocked by a call to `notify`
- Wait does not return until the waiting thread acquires the object's monitor.

`wait()` is really a convenience method. It saves you the trouble of having to explicitly release and reacquire the lock.

The `java.util.concurrent.lock` package has a `Condition` class that performs a similar function. But `Condition` gives you more flexible usage patterns.

Here's a home-grown example of using `Condition` objects.

```
final ReentrantLock lock = new ReentrantLock();
final Condition condition = lock.newCondition();
private boolean state = false;

public void checkState() throws InterruptedException {
    lock.lock();
    try {
        condition.await();
System.out.println("current state = " + state);
    }
    finally {
        lock.unlock();
    }
```

```
}

public  void changeState() throws InterruptedException {
    lock.lock();
    try {
        this.state = (! this.state);
        condition.signal();
    }
    finally {
        lock.unlock();
    }
}
```

Like `Object.wait()`, `Condition.await()` implicitly releases the lock until another thread calls `signal()` on the same condition object. When `await()` returns, the calling thread is guaranteed to have reacquired the lock.

In addition to `signal` (wakes up a single thread), there's also a `signalAll` that wakes up all awaiting threads.

If several threads are waiting on a single condition, the decision of which thread to wake up is made in a JVM-specific manner. (i.e. - consider it a nondeterministic decision, and expect the results to vary with different JVM implementations).

2.6.2 Thread Synchronization Tools

Among the tools provided by the `java.util.concurrent` package are

- locks, like `ReentrantLock`. *Mutex* is another name for a lock.
- read/write locks, like `ReentrantReadWriteLock`.
- Semaphores
- Barriers

2.6.3 Read/Write Locks

Reentrant locks allow one thread to access a critical section (or shared data) at any given time. In some cases, this restriction can cause an unnecessary reduction in concurrency. Suppose we have a set of shared data that is read often, but only written occassionally. We'd prefer to allow concurrent reads, but exclusive writes.

`ReentrantReadWriteLock` and its inner classes `ReentrantReadWriteLock.ReadLock` and `ReentrantReadWriteLock.WriteLock` provide this service.

You create a single `ReadWriteLock`, and use that to obtain instances of the inner classes.

Here's an example from the javadoc:

```
private final Map<String, Data>  m = new TreeMap<String, Data>();
private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
private final Lock r = rwl.readLock();
private final Lock w = rwl.writeLock();

/* shared access, use read lock */
public Data get(String key) {
    r.lock(); try { return m.get(key); } finally { r.unlock(); }
}
```

```
/* exclusive access, use x-lock */
public Data put(String key, Data value) {
    w.lock(); try { return m.put(key, value); } finally { w.unlock(); }
}
```

You can acquire a read lock any time a write lock is not held. On the other hand, write locks are exclusive.

As expected, the compatiblity matrix for read and writes locks is

	Read	**Write**
Read	Y	N
Write	N	N

When are read/write locks appropriate?

- If there are many readers.
- If reads occur more frequently than writes.
- If reads take longer than writes.

2.6.4 Next Class

In our next class, we'll cover semaphores and barriers.

2.7 Lecture – 10/16/2007

2.7.1 Semaphores

Java's semaphore implementation is `java.util.concurrent.Semaphore`.

Semaphores are like locks. They can be used to govern or restrict access to shared data. Unlike locks, semaphores have a value associated with them. We refer to this value as the *number of permits*. Acquiring the semaphore reduces the number of available permits. Releasing the semaphore increases the number of available permits.

In Java, the number of permits is set in the semaphore's constructor.

The most important Semaphore methods are `acquire` and `release`.

`acquire` grants one or more permits, reducing the number of permits available. If no permits are available, then `acquire` blocks until the requested number of permits is available.

`release` returns one or more permits.

A semaphore with one permit behaves very much like mutex.

Semaphore can be used to simulate read/write locks. A "read lock" will acquire one permit; a "write lock" will acquire all permits.

Semaphores are commonly used in pooling applications. Suppose we wish to write a connection pool class. The maximum number of connections is bounded. Checking out a connection acquires a permit; returning a connection releases a permit. The basic idea is shown in Figure 2.7.

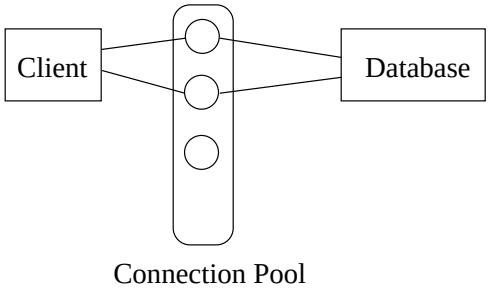

Figure 2.7: Diagram of A Connection Pool

2.7.2 Semaphores vs. Locks

Semaphores have no condition variables; locks do.

Locks are re-entrant. Semaphores are not. For example, this code will work for a single thread:

```
ReentrantLock mLock = new ReentrantLock();
public int fact(int x) {
    try {
        mLock.lock();
        System.out.println(mLock.getHoldCount());
        if (x <= 0) {
            return 1;
        }
        return x * (fact(x - 1));
    }
```

```
        finally {
            mLock.unlock();
            System.out.println(mLock.getHoldCount());
        }
    }
```

However, this code will block at 5 recursive calls, because the recursion exhausts the number of permits.

```
    private Semaphore sem = new Semaphore(5);
        public int fact(int x) {
        try {
            sem.acquire();
            System.out.println(sem.availablePermits());
            if (x <= 0) {
                return 1;
            }
            return x * (fact(x - 1));
        }
        catch (InterruptedException e) {
            return 0;
        }
        finally {
            sem.release();
            System.out.println(sem.availablePermits());
        }
    }
```

Locks have a notion of ownership; semaphores do not. If thread t_1 acquires a lock, then t_1 must release it. However, if thread t_1 acquires a permit from a semaphore, it's okay for another thread t_2 to release the permit.

General rules of thumb:

- If your shared data consists of a single object, locks are probably the most appropriate concurrency control mechanism.
- If your shared data consists of a set of distinct objects, semaphores may be more appropriate.

2.7.3 Barriers

A barrier is like a rendezvous point for multiple threads. All threads must meet at the barrier before any of them are allowed to pass through the barrier.

Java's barrier is a `java.util.concurrent.CyclicBarrier`. The constructor sets the number of threads participating in the Rendezvous.

Barrier's are like `join()`, but threads can continue after the barrier is reached.

Here's my own (contrived) barrier example. In this case, each thread hits the barrier 5 times.

```
    import java.util.concurrent.BrokenBarrierException;
    import java.util.concurrent.CyclicBarrier;

    public class Barrier3 {
        static class Worker implements Runnable {
            CyclicBarrier b;
            int jobs = 0;
            int value = 0;
```

```
        public Worker(CyclicBarrier b, int count) {
            this.b = b;
            this.jobs = count;
        }

        public int getValue() {
            return value;
        }

        public void run() {
            try {
                for (int iter = 0; iter < jobs; iter++) {
                    value++;
                    Thread.sleep(500);
                    b.await();
                }
            }
            catch (InterruptedException e) {
                return;
            }
            catch (BrokenBarrierException e) {
                return;
            }
        }
    }

    static class Collector implements Runnable {
        Worker workers[];
        public Collector(Worker w[]) {
            this.workers = w;
        }
        public void run() {
            for (int iter = 0; iter < workers.length; iter++) {
                System.out.print(workers[iter].getValue());
            }
            System.out.println();
        }
    }

    public static void main(String argv[]) {
        Worker w[] = new Worker[3];
        Collector c = new Collector(w);
        CyclicBarrier b = new CyclicBarrier(3, c);
        for (int iter = 0; iter < w.length; iter++) {
            w[iter] = new Worker(b, 5);
        }

        for (int iter = 0; iter < w.length; iter++) {
            Thread t = new Thread(w[iter]);
            t.start();
        }
    }
}
```

2.7.4 Barriers and Join

`join` is a very simple form of barrier. In this case, the rendezvous point is the end of thread execution. Barriers give you more flexibility. You can put the barrier in the middle of `run`; you can also set several barriers in `run`.

Threads that participate in the rendezvous call `barrier.await()`.

CyclicBarrier's constructor takes an optional runnable object. If provided, this Runnable is run when the barrier is tripped, but before the threads are allowed to continue.

2.7.5 The Strategy Pattern

We'll build a file cache for one of our next homework assignments. The cache has to implement six different cache/locking strategies. The basic strategy pattern is shown in Figure 2.8.

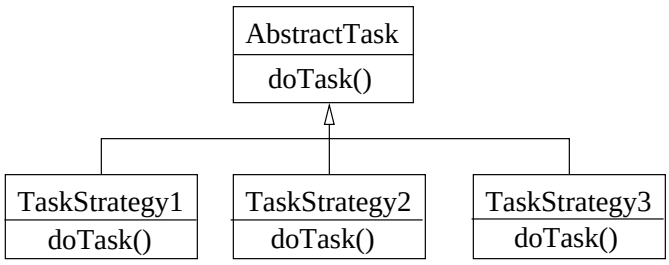

Figure 2.8: Strategy Pattern

In Figure 2.8, we have a basic object called `AbstractTask`, and three different implementations. Each implementation (say) performs the same function, but uses a different algorithm to do it.

Strategies are often constructed from Factories. We might have something like this:

```
public AbstractTask factory() {
  AbstractTask t;
  if (isRaining()) {
    t = new TaskStrategy1();
  }
  else if (isSunny()) {
    t = new TaskStrategy2();
  }
  else {
    t = new TaskStrategy3();
  }
  return t;
}
```

This kind of approach allows us to choose an optimal strategy for a given task.

For our project, the base class will be called `FileCache`, and the six strategies will be

- LRUFileCacheMutex
- LRUFileCacheRWLock
- LRUFileCacheSemaphore
- LFUFileCacheMutex
- LFUFileCacheRWLock
- LFUFileCacheSemaphore

As one would expect, LRU is *least-recently used* and LFU is *least-frequently used*.

We have complete freedom in how the class hierarchy for these strategies is organized. We can use six independent implementations, or we can do something fancier.

Our FileCache will have ten cache slots. Our functional requirement is to get ten slots working with twenty files, where each file is some web page (save the web pages to a directory).

This assignment will be due on 10/30/2007.

2.8 Lecture – 10/19/2007

2.8.1 Futures

A future is

> an agreement or an organized exchange to buy or sell assets (commodities or shares) at a fixed price, but to be delivered and paid for later.

In concurrent programming, Futures provide an alternative to synchronous methods calls. In a synchronous method call, client code that calls `foo()` will block until `foo()` returns. Futures all the client code to continue executing, in parallel with `foo()`. Figure 2.9 shows flow control for a synchronous method call.

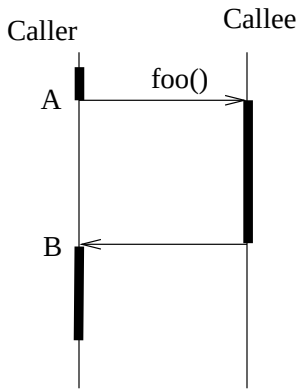

Figure 2.9: Diagram of a Synchronous Method call

In Figure 2.9, point A marks the moment in time that `foo()` is called, and point B marks `foo`'s return. By using a future, the caller could continue to execute between A and B.

2.8.2 Proxy Design Pattern

The proxy design pattern allows us to use a surrogate (or placeholder) object in place of a real one.

For example, consider image loading in a web browser. Images are embedded in an html page, usually tagged with their width and height. When you first load the page, the browser fills in the image area with a bounding box, and then fetches the images in the background. Once the images have been downloaded, the bounding box is replaced with the real image. In this example, the bounding box acts as a proxy for the real image.

Figure 2.10 shows a UML diagram of the proxy pattern.

In Figure 2.10, an "extent" provides information about the image's size. An ImageProxy gets extent information from the HTML tags, while the Image gets extent information from the actual image file.

There is always a 1:1 relationship between the proxy object and the real object.

Here are a few examples of ImageProxy's code:

```
void ImageProxy::draw() {
  if (image == NULL) {
    drawBBox();
  }
  else {
```

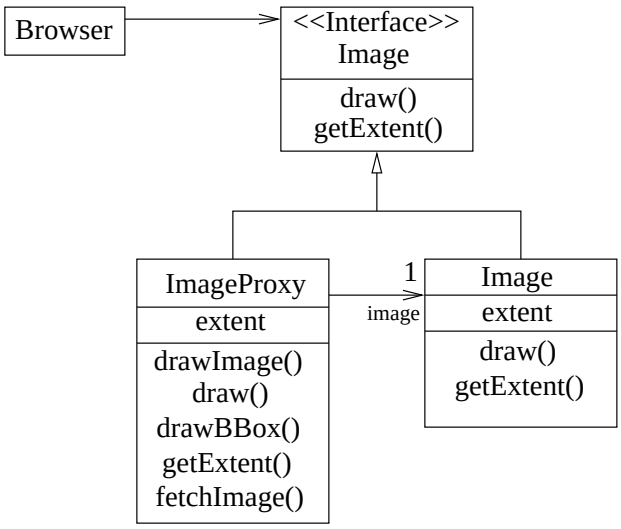

Figure 2.10: UML Diagram of Proxy Pattern

```
    image->draw();
  }
}

Extent ImageProxy::getExtent() {
  if (image == NULL) {
    return extent;
  }
  else {
    return image->getExtent();
  }
}
```

2.8.3 Futures and the Proxy Pattern

The proxy pattern is usually used to implement futures. Calling a future method gives you a "receipt" to obtain the real object later. The receipt object acts like a proxy. When the future method completes, the future will give you the real object.

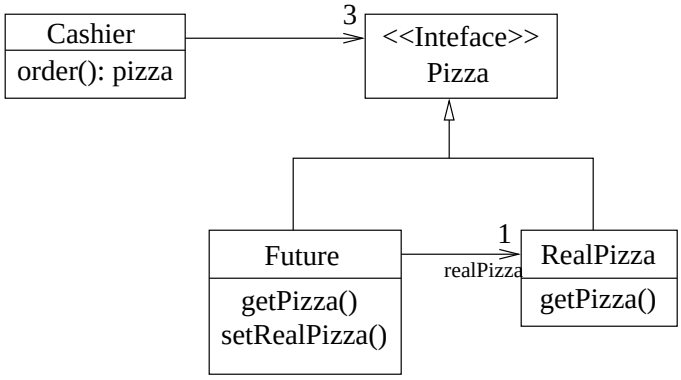

Figure 2.11: UML Diagram of Future

Figure 2.11 shows a UML diagram of a future (this particular one is the "pizza" example we discussed in class).

Future can be thought of as a design pattern. Futures are also referred to as asynchronous method invocations.

2.8.4 More on Method Invocation Patterns

Figure 2.9 showed a diagram of a synchronous method invocation. There are three other invocation patterns we're going to study.

Figure 2.12 shows an example of Future invocation, also referred to as asynchronous method invocation. After `foo()` is called, both the caller and callee (Future) continue to execute. Between points A and B, we assume that the caller periodically checks to see if the Future has completed. The caller must (eventually) retrieve the real object from the future.

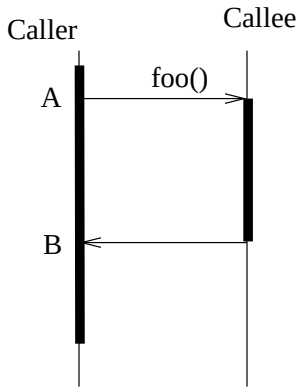

Figure 2.12: Future Method Invocation

Figure **??** illustrates a callback. It's similar to a future: both the caller and callee continue to execute. The difference lies in how the return value is communicated from the callee to the caller.

In a Future, the caller retrieves the real value from the callee. With a callback, the callee invokes a method on the caller object when the callee has finished. (The caller must supply the callback method to the callee).

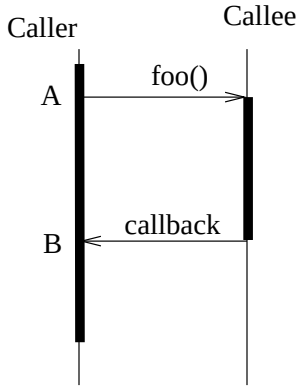

Figure 2.13: Callback Method Invocation

Figure 2.14 shows a one-way method invocation. No value is returned to the caller. The caller invokes the callee and continues to execute. The callee performs its job and then exits.

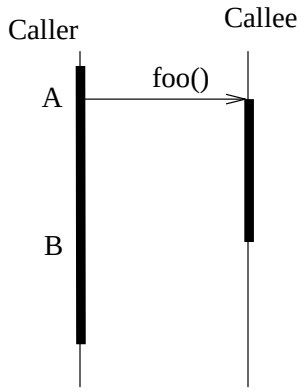

Figure 2.14: One-way Method Invocation

When working with Futures and Callbacks, we have to be careful of shared data. Because two threads are executing concurrently, shared data can become corrupt unless we take appropriate precautions (i.e. - use locks).

2.8.5 HW18

For hw18, we're going to make several modifications to the Future example discussed in class. This includes (but is not limited to)

- Implementing a `Future.isReady()` method that tells the caller whether the future has finished. We'll need to modify the caller code to use `Future.isReady()`. If the Future is not ready, the caller should perform some visible work until the Future becomes ready.
- Add an overloaded method `Future.getPizza(long timeout)`. This behaves like the existing `Future.getPizza()`, but throws an exception if the pizza is not ready before the timeout occurs.

This assignment will be due on 11/1/2007.

2.9 Lecture – 10/23/2007

2.9.1 Mediator Design Pattern

The mediator design pattern encapsulates the interaction of a set of objects. This pattern allows the interaction to be controlled in one place, and it also promotes loose coupling among the objects involved.

Figure 2.15 shows the basic mediator pattern. (Italic names denote interfaces or abstract classes.)

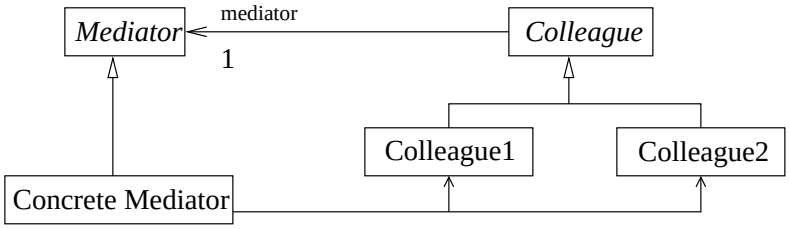

Figure 2.15: Mediator Design Pattern

A ConcreteMediator controls the actions between colleagues. For example, we might use this pattern to design an order processing system. In an order processing system, the Colleague objects would be classes like Manager, Order, and Account. A reference to each Colleague is passed to the ConcreteMediator to conduct the interaction.

If you have an application with a lot of business rules, the mediator pattern will help you to consolidate the business logic in one place. (I guess we can also call the mediator a Controller).

It's also possible for the mediator to refer directly to instances of the Colleague interface. When the Mediator refers to the Colleague interface (and not the concrete Colleague classes), then Colleague is really acting like a strategy.

2.9.2 Producer/Consumer Pattern

Producer/Consumer is a basic pattern in concurrent programming.

The *producer* is a thread (or group of threads) that generates data. Generated data is placed on a queue (or similar form of shared data structure).

The *consumer* is a thread (or group of threads) that removes data from the shared queue and processes it.

In general, when a consumer tries to retrieve data from an empty queue, then the consumer will block until the producer makes data available.

Of course, because the queue is a shared data structure that's accessed by multiple threads, one needs to put the appropriate mutual exclusion guards in place.

2.9.3 Homework 19

- Read chapter on Mediator design pattern
- Implement a producer/consumer program. For a producer, we will use the Fibonacci number generator that we wrote in an earlier assignment. The producer will place data in a *channel*.
- We will need to write a consumer that removes data from the channel. If data is not available, the consumer must wait for it. Use locks and condition objects to control the synchronization.

- The structure of the channel object is shown in Figure 2.16. Of course, the producer will call `put()` and the consumer will call `get()`.

  The Channel should behave like a mediator – by implementing mutual exclusion, the channel will control how the producer and consumer interact.

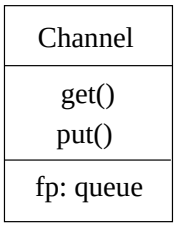

Figure 2.16: Structure of Channel Object

Futures are also similar to Mediators. Using our Pizza example, `Order` is the producer.

2.10 Lecture – 10/25/2007

2.10.1 Java's `synchronized` keyword

In java, declaring a method as `synchronized` implicitly creates an critical section that contains the entire method body. For example, the following two code fragments are equivalent:

```
public synchronized void foo() { ... }

public void foo() {
  // Assume aLock is a ReentrantLock class member
  aLock.lock();
   ...
  aLock.unlock()
}
```

The scope of `synchronized` is always per-method, or per synchronized block. Each object contains its own mutex (as described by the java language specification). That's what synchronized is locking.

ReentrantLocks have arbitrary scope. ReentrantLocks also allow one to define multiple locks per object.

2.10.2 Concurrency: A Double-Edged Sword

When we use locks to protect shared resources, we must identify:

- What must be protected
- Who accesses the shared resources
- Where are shared resources changed.

As a general rule of thumb, you should be pessimistic when restricting access to shared resources (you don't want data corruption). However, too much synchronization is bad too. If you try to synchronize everything, you'll

1. cause a loss of concurrency, and
2. cause the JVM to expend excessive resources acquiring and releasing locks.

In Java, lock acquisition and release is not cheap. So, for performance, it's best to minimize the use of locks and thread synchronization.

2.10.3 Minimizing Threading Errors

One can reduce threading errors by using *immutable* objects. Immutable objects cannot change state – they have no setter methods, only getters. Because the object cannot change state, there's no opportunity for race conditions. `java.lang.String` is an example of an immutable object. The primitive type wrapper classes (`Integer`, `Boolean`, etc) are also immutable.

Within `String`, member variable that holds character data is `private` and `final`. Once you create a string, it's impossible to change. In addition, the `String` class is declared final, so one can't change its behavior by subclassing.

`final` variables can be assigned in one of two ways:

- When the variable is declared
- Inside the Class's constructor

Data becomes *mutable* when you define a setter method; this allows callers to change the data.

To make data immutable:

- Use `private` and `final` as often as possible. The compiler will detect any illegal accesses.
- Remember that small changes can cause race conditions.
- Document immutability (in design documents, and API documents). For UML, use the {`frozen`} qualifier.

2.10.4 Common Pitfalls of Immutable Object Design

Be wary of pointers that are declared final. In java, this means that you can't reassign the pointer; it does *not* prevent you from changing the object that the pointer references. For example

```
public class Mutable {
  private final ArrayList<String> data = new ArrayList<String>();

  public Mutable(String s1, String s2) {
     data.add(s1);
     data.add(s2);
  }

  public ArrayList<String> getData() {
    return data;
  }
}
```

This class is not immutable. It gives out a reference to it's `data` member. Any outside code that calls `getData()` can change the contents of the ArrayList.

Another example:

```
public class Mutable2 {
  private final Point p;

  public Mutable2(Point p) {
     this.p = p;
  }
}
```

`Mutable2` stores a reference to the caller's Point object. The caller can still change the data values inside the Point.

Primitive fields are not subject to these pitfalls. Once you set them, they cannot change.

2.10.5 Minimizing Performance Loss

If appropriate, use separate read and write locks to synchronize access to shared data. This is an appropriate strategy when shared data will be read often, but changed infrequently. With read and write locks, readers will not have to compete with one another.

If you have only reader threads, use Immutable objects.

If you have only writer threads, use the `volatile` keyword. (We'll discuss this in detail during our next lecture).

Use thread specific storage.

2.10.6 Performance Implications of Immutable Objects

Immutable objects can improve performance. There's no need for locking or data synchronization.

However, most objects have getter and setter methods. In some cases, it's beneficial to make two version of a class: a Mutable one and an Immutable one.

`String` and `StringBuffer` are examples of this approach. `StringBuffer` is mutable; `String` is not. `StringBuffer` uses synchronization to prevent concurrent modification to its internal data. (Of course, there's a third version, `StringBuilder`, which is an unsynchronized version of StringBuffer.).

2.10.7 Homework 20

We'll write two classes: `MutablePerson` and `Person` (the latter of which is Immutable). `MutablePerson` should use read and write locks to synchronize data access, much like StringBuffer does.

`ImmutablePerson` should use minimal synchronization (as little as possible). (No more than once).

This is due 11/8/2007.

2.11 Lecture – 11/1/2007

2.11.1 Java's volatile keyword

In some (specific) cases, you can use the `volatile` keyword to avoid the need for locking. `volatile` is useful when

- There is not a need to synchronize threads
- There is a need to synchronize memory.

A common example of this is a *latch* (or *flag*).

Note that we are talking about two different types of synchronization here: *thread synchronization* and *memory synchronization*. `volatile` does *not* help with thread synchronization; it's only useful for memory synchronization.

2.11.2 Java's Memory Model

Java recognizes two different types of memory: *working memory*, which is "owned" by individual threads, and *main memory*, which is global to the JVM.

When a thread T reads variable v, v's value is copied from main memory to T's working memory. T uses the value from working memory. This pattern is called *Read-Use*: v is read from main memory, and T *uses* v from working memory. This is illustrated in Figure 2.17.

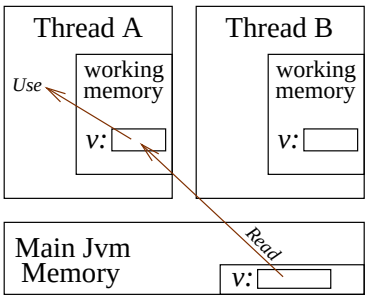

Figure 2.17: JVM Memory Model

Read and Use are primitive operations, defined by the Java Virtual Machine specification.

If a variable v is not in working memory, then v's value must be accessed using "Read-Use". However, if v is resident in T's working memory, then T has two options: *use* the copy of v that's in working memory, or use Read-Use to take v's value from main memory. The JVM specification is (intentionally) vague in this area; the choice is really up to JVM implementers.

A JVM vendor would probably choose Use when v is resident in working memory (faster than Read-Use); but, it would also be correct for a vendor to utilize Read-Use for each variable access.

Of course, if T issues the operations Read, Use, Use, Use, ..., then it is possible for (1) the main memory value of v to change and (2) for T to use an old value of v for a (short) period of time.

New values are written to memory in a two step process. When a thread T changes the value of a variable v, T will *Assign* v in its working memory, and eventually the JVM will *Write* v back to main memory. The pattern here is "Assign-Write".

Write occurs after Assign, but the interval between Assign and Write is not specified. If T changes v multiple times, the pattern could be Assign-Write, Assign-Write, Assign-Write; or, it could be Assign, Assign, Assign-Write.

Like Read and Use, Assign and Write are JVM primitive operations.

Like Read-Use vs Use, Assign-Write vs Assign is up to the JVM Implementation.

According to the JVM specification, Assign, Write, Read, Use, Lock, and Unlock are all atomic, uninterruptible operations. Thread context switches cannot occur in the middle of these operations.

2.11.3 Lock and Unlock

The bodies of `synchronized` methods are implicitly surrounded with locks (taken on the `this` object).

```
public synchronized void foo() {
   // critical section
}
```

and

```
public void foo() {
  lock.lock();
  // critical section
  lock.unlock();
}
```

behave the same way.

The JVM performs two actions around critical sections: thread synchronization and memory synchronization.

Locks & Thread Synchronization. Only one thread can execute code in a critical section; all other threads must wait.

Locks & Memory Synchronization. After a thread enters a critical section, it must flush working memory to main memory, and destroy working memory. Upon exiting a critical section, working memory must be flushed to main memory.

Thread unsafety results from the failure to synchronize threads, or from the failure to synchronize memory.

2.11.4 Volatile Variables

Locks and `synchronized` allow us to control thread synchronization.

The `volatile` keyword allows us to control memory synchronization. Any Use of a volatile variable immediately causes a Read. Any Assign to a volatile variable immediately causes a Write.

`volatile` does only memory synchronization; *not* thread synchronization.

With volatile, memory synchronization occurs automatically. If you only need to ensure that each thread sees the newest value of a variable (say, for a flag or latch), then declaring the latch `volatile` can eliminate the need for locking. Of course, by avoiding locking, we also reduce overhead.

`volatile` is best used with primitive types. If an object field is declared `volatile`, then the pointer is affected; *not* the object pointed to.

`volatile` works for single read/write operations. For example, the results of these operations are memory synchronized:

```
volatile int a;
a = 0;
if (a == 0) { ... }
```

Intermediate operations on a volatile variable are *not* memory synchronized (i.e., intermediate results on computations involving volatile variables are not themselves volatile). Neither of these is safe:

```
volatile in a;
a = a + 1;
a++;
```

Do not use volatile with array variables. The pointer to the array will be volatile; the contents of the array will not.

2.11.5 Singleton Design Pattern

The Singleton pattern is used to guarantee that a class has only one instance. The general form is:

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { ... }

  public static synchronized Singleton getInstance() {
       if (instance == null) {
          instance = new Singleton();
       }
       return instance;
  }
}
```

The example above works correctly in Java (no race conditions).

2.11.6 Double Checked Locking Pattern

One might be tempted to use double-checked locking (also known as *test-test-and-set*, or DCLP) to avoid locks after the singleton has been instantiated.

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { }

  public static Singleton getInstance() {
       if (instance == null) {
          lock.lock();
          if (instance == null) {
             instance = new Singleton();
          }
          lock.unlock();
       }
       return instance;
  }
```

In java, this **does not work**. The JVM's memory model causes a race condition: fields initialized by the singleton's constructor might not be immediately visible to other threads.

In other languages (like C++ ... or anything without a two-copy memory model), DCLP works correctly.

Static initializers can also create singletons safely:

```
public class Singleton {
  private static Singleton instance = new Singleton();
```

```
  private Singleton() { }

  public static Singleton getInstance() {
     return instance;
  }
}
```

2.12 Lecture – 11/6/2007

2.12.1 Thread-Specific Storage

Thread-specific storage consists of one or more variables that are associated with a single thread T. T's thread-specific storage is visible only to T, and not to any other thread T'.

In java, thread-specific storage is implemented by the class `java.lang.ThreadLocal`.

Consider: we have a set of threads $T_1 \ldots T_n$ that need to manipulate variables (in a larger scope than a method call). We could hold all of these variables is a global container (say, a List). However, if we took this approach, we'd need to synchronize access to the list.

Now consider an alternative approach. Our global contain is a table, each thread "owns" one table slot, and each thread accesses *only* the slot that it owns. This is shown in Figure 2.18.

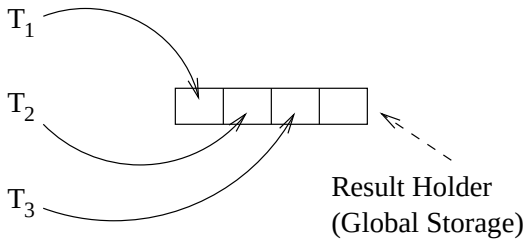

Figure 2.18: Global Table: each thread owns and accesses one slot

In Figure 2.18, there is no need for synchronization; the slots accessed by T do not overlap with the slots accessed by T'. Because there is no need for synchronization, this results in safer code.

Figure 2.18 is how `ThreadLocal` is implemented. The class uses a map of thread $\mapsto$ value. ThreadLocal does its own (internal) concurrency control where needed, and each thread can access only the data that it owns.

Part 3

Networking and IPC

3.1 Lecture – 11/6/2007

In this part of the course, we'll cover networking and inter-process communication (IPC). IPC allows us to develop a set of APIs that can handle (1) communications between two processes running on the same machine and (2) communications between two processes running on different machines. With properly designed methods, the difference between (1) and (2) can be very small.

Figure 3.1 shows an illustration of (1), and Figure 3.1 shows an illustration of (2).

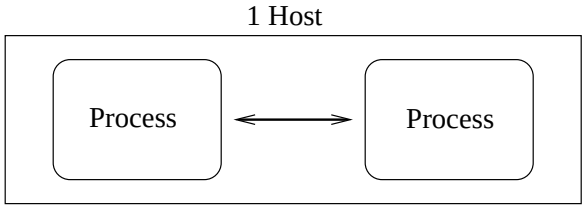

Figure 3.1: IPC: two processes on the same machine

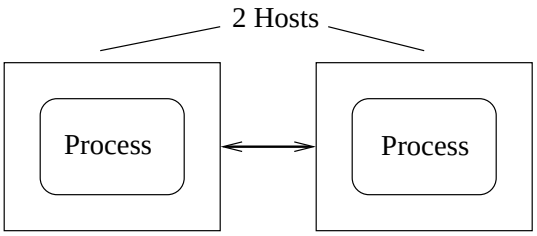

Figure 3.2: IPC: two processes on different machines

3.1.1 Protocol Stacks

A network protocol stack is divided into layers. Each layer handles a specific aspect of networking communications. For example:

- Application Layer (HTTP, POP)
- Session Layer (SSL)
- Transport Layer (TCP, UDP)

- Network Layer (IP)
- MAC Layer (Mac Addresses)
- Physical layer (Ethernet, FDDI, etc)

(I suppose this is a simplification of the standard OSI model – the OSI model has seven layers, and we've shown fewer than that).

A *protocol* allows two (or more) processes to communicate with each other in an unambiguous way. Each protocol defines:

1. The messaging primitives and commands
2. Request/Response pairs
3. A message format.

For example, HTTP and POP3 are protocols.

3.1.2 The Socket Interface

Sockets facilities are provided by the operating system call interface. Sockets originated with BSD in 1984. These days, just about every operating systems vendor provides a BSD-compatible socket library.

The socket API implements the transport layer. Common socket methods include:

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `read()`, `write()`, `select()`, `close()`

A socket is a communications channel for the transmission of TCP and UDP data between processes. The communicating processes could reside on the same machine, or on different machines.

Simplistically, the client will call `socket`, then `connect`, then `write`. The server will call `socket`, then (bind and) `listen()`, and `read()`. The basic idea is illustrated in Figure 3.3.

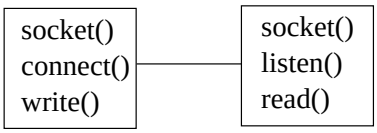

Figure 3.3: Simple view of socket communications

TCP is full-duplex (unlike pipes). Both client and server can read from and write to a single socket connection.

At the operating system level, sockets are referenced by file descriptors. File descriptors, in turn, reference kernel data structures.

File descriptors can be used to represent pipes, sockets, regular files, special devices, and so fourth.

Most operating systems define three "standard" file descriptors. (In unix, every process has these)

fd 0 standard input
fd 1 standard output
fd 2 standard error

Note that these also constitute a form of IPC. The allow a command to communicate with the shell that invoked it.

3.1.3 Java Network API

Java's networking API lives in the package `java.net`. These APIs implement TCP and UDP communications. They're essentially glue to the OS networking facilities. Java tends to follow the BSD interface rather closely.

3.1.4 IP Addresses and Port Numbers

One references a remote process using a combination of IP address and port number. For example, www.cs.umb.edu:80.

Process ids are not suitable for identifying remote processes. (The process ids change each time the machine is rebooted, and each time the process is restarted).

IP address uniquely identifies a machine; port numbers uniquely identify a process running on a single machine.

Well-known protocols have standard port numbers. When writing a new network service, you should avoid using well-known ports.

3.1.5 Java Sockets

This is a brief outline of how a server process uses the java socket API:

```
ServerSocket serverSocket = new ServerSocket(9000);
Socket socket = serverSocket.accept();
Scanner scanner = new Scanner(socket.getInputStream);
PrintWriter writer = new PrintWriter(socket.getOutputStream);
```

And here is a brief outline of the client calls:

```
Socket socket = new Socket("localhost", 9000);
Scanner scanner = new Scanner(socket.getInputStream);
PrintWriter writer = new PrintWriter(socket.getOutputStream);
```

3.1.6 HW 21

Read papers 5 & 6 on the course website. There's nothing to turn in for this assignment; just read the papers.

3.1.7 HW 22

We'll be extending the Simple Bank Protocol program.

- Have the bank server maintain multiple accounts (use a `Collections` object to hold them)
- Extend the protocol, so that all operations take an account number. (See homework handout for details).
- Make the BankAccount class thread safe, by using read and write locks.
- Have each request handled by a separate thread.
- At your option - write a thread pool (not required for this assignment).
- Due 11/20/2007.

3.2 Lecture – 11/10/2007

3.2.1 Code Reuse in OOP

Three common ways to re-use object oriented code:

- Inherit from an existing class. For example SSLServerSocket is-a ServerSocket
- Use (subclass) an existing class. For example, subclassing Thread.
- Combine instances at runtime. For example, Java's IO framework.

Java IO relies heavily on the decorator pattern.

3.2.2 Decorator Pattern

Suppose you have a set of variations of behavior S, but many combinations of these variations. We *could* write one class per variation, but we'd like to do better than writing $\mathcal{P}(S)$ different classes.

Decorator works well with this kind of scenario. Decorator allows us to write small pieces of functionality that can be strung together in different ways.

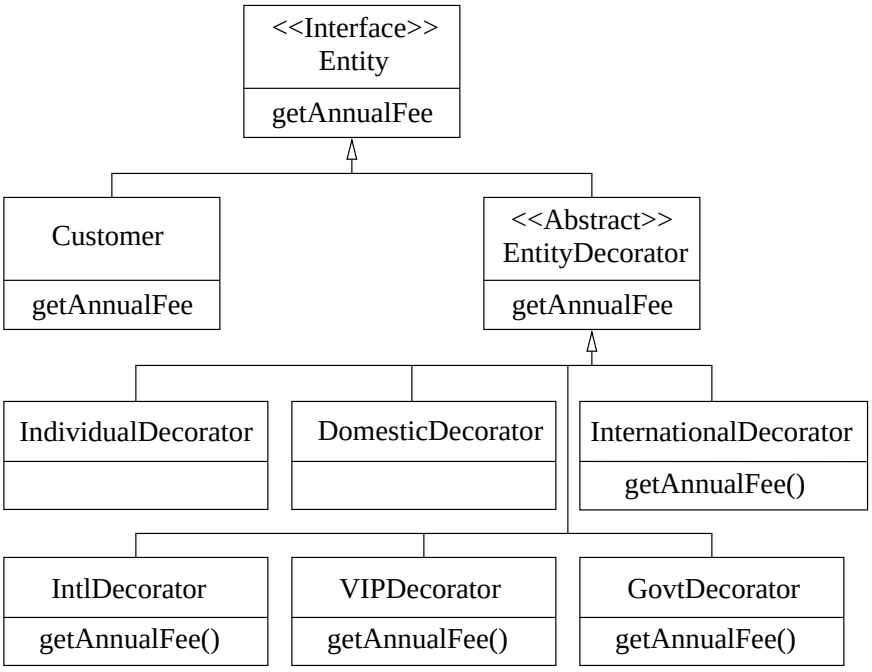

Figure 3.4: Example of Decorator Pattern

In Figure 3.4,

- Customer is a concrete class (what we are going to decorate)
- EntityDecorator is the parent class of all decorator objects
- Entity is the parent class of both Customer and EntityDecorator. Note that Entities and Decorators share the same interface.
- The other six classes are decorators.

With this class structure, we can combine decorators to compute annual fees of different customer types. Figures 3.5 and 3.6 show two different combinations of decorators.

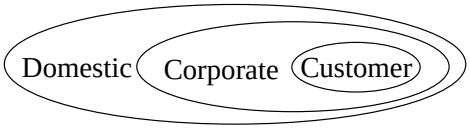

Figure 3.5: One Combination of Decorators

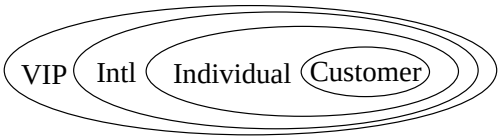

Figure 3.6: Another Combination of Decorators

Decorators allow you to string different pieces of functionality together. Each decorator maintains a reference to the object it "contains". Figure 3.7 shows the flow of information information in the decorator pattern.

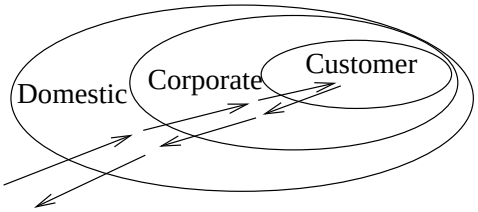

Figure 3.7: Information Flow In Decorator Pattern

3.2.3 Java's IO Package

Java divides IO into two categories: byte IO and character IO. `InputStream` and `OutputStream` are the base byte IO classes, and `Reader` and `Writer` are the base classes for character IO.

`FilterInputStream` is the base class for IO input filters. For example:

- `GzipInputStream`
- `BufferedInputStream`
- `CipherInputStream`

`FilterOutputStream` is the base class for IO output filters.

Java also provides Writer and Utility classes:

- `PrintWriter`
- `Scanner` (a simple string parser)
- `FileReader` and `FileWriter`

3.2.4 For Next Class

Look over `java.nio` classes.

3.3 Notes from *A Fundamental Turn in Concurrency in Software*

Article published in Dr. Dobb's Journal, March 2005.

- For many years, we've enjoyed a "free lunch". CPUs would get faster every year, through higher clock speeds, instruction optimization, and cache. This enabled all programs to run more quickly. It also made it easier to write programs that ran "fast enough" without having to give much thought to optimization.
- If one were to plot average CPU speeds over time, a trendline would indicate that we should be using 10GHz+ CPUs. However, there aren't even 4GHz CPUs commercially available. (Intel had plans for a 4GHz Pentium, and then wound up cancelling the project.)
- Increases in straight-line execution will no longer be able to provide the performance improvements predicted by Moore's law. CPUs continue to become capable of doing more work (i.e., by using multiple cores), but they are not getting any *faster*.
- In the near-term future, there will be three main drivers of hardware performance growth: hyper-threading, multicore, and cache. The first two rely on programmers to build concurrency into their applications.
- Space is speed. By going from a 32-bit CPU to a 64-bit CPU you may gain more registers, and a wider instruction line. However, any pointers in your program are now twice as big – that causes cache misses, which in turn slows your program down.
- OOP was the great programming revolution of the 1990's. The next great revolution will involve concurrency and multi-threading.
- "You can only support a software development revolution on a technology that's mature enough to build on ... and it generally takes at least seven years before it's solid enough to be broadly usable without performance cliffs and other gotchas."
- Where is multi-threading beneficial? (1) to separate logically different execution flows, and (2) to take advantage of multiple cores, or multiple CPUs.
- The key to good performance with multi-threaded code: finding sensible ways to parallelize operations, and minimizing the use of shared state (i.e., minimize the use of locking).
- Concurrency is much harder to reason about than straight-line execution.
- In order to learn a new paradigm, you need to (1) understand the correct designs and (2) understand why they're correct.
- To ways to deal with the concurrency sea change: (1) redesign your applications for concurrency and (2) write code that is more efficient and less wasteful.
- Implicitly parallelizing compilers can help, but don't expect much help.

3.4 Notes from *Software and the Concurrency Revolution*

Article published in ACM Queue, September 2005.

- Computers will continue to become more and more capable, but programs can no longer simply ride the hardware wave of increasing performance unless they are highly concurrent.
- Concurrency is hard. Today's tools are inadequate for transforming programs into parallel applications; it is difficult to find opportunities for parallelization in mainstream applications; and, concurrency requires programmers to reason about their programs in ways that humans find difficult.
- If concurrency is integral to performance, languages that offer native support for concurrent programming concepts will continue to thrive. Those that offer little/no support for concurrent programming concepts will fade from popularity.
- For many server-based applications, concurrency is a "solved problem" (i.e., most server applications already handle many client requests in parallel). With server applications, there tends to be little sharing of data among client requests.

  Client and desktop applications are another story. There's typically a lot of shared state.
- The smaller a work unit is, the more you have to think about "does the cost of spawning new threads to divide the work overshadow the benefit of doing the work in parallel?".
- *Independent Parallelism.* All sub-tasks can be performed completely independently. There are no interactions between sub-tasks, and no sharing of data (except, perhaps, at the end of the computation).

  Example, given `double A[100][100]`, it's very straightforward to parallelize `A = A * 2`.
- *Regular Parallelism.* Operations on data are mutually dependant, but in a predictable way. For example `A[i,j] = (A[i+1, j] + A[i-1, j] + A[i, j+1] + A[i, j-i])/4`. (A smoothing filter).

  To parallelize this computation (and get all of the data dependencies right), you'd have to (1) make a new copy of the data or (2) traverse the array in passes of diagonals.
- *Unstructured Parallelism.* Data accesses are not predictable, and need to be coordinated through explicit synchronization.

  Unlike many other facets of programming, lock-based code is not composable. For example, if you have correct libraries A and B, you can use both in the same program, and they'll probably work correctly. On the other hand, two units of lock-based code may not work properly when combined; both units may acquire locks L_1 and L_2, but they do so in opposite orders, causing deadlock.

  Practices like lock hierarchies can work well in individual frameworks, but are hard to make reusable. You can't predict the order in which the caller will attempt to access resources.
- Three problems with Java's `synchronized` keyword:

  1. Not appropriate for code that calls virtual functions. If C holds a lock while calling a method in subclass C', we have the possibility for deadlock.
  2. `synchronized` can cause too much locking. It affects all instances; even those that are never shared among threads.
  3. `synchronized` is often the wrong grain of locking. By itself, `synchronized` cannot guarantee consistency across a set of method calls. If the need for atomicity does not correspond to a method call boundary, then you need to use explicit locking anyway.
- Two alternatives to locking: transactional memory and lock-free programming. The former treats memory as a database treats rows (two-phased locking of memory locations within sections of code

marked as atomic); the latter takes advantage of data structures that requires no locking. Both are still in the research stages.

- What we need are (a) new languages that offer more support for concurrency or (b) new extensions to existing languages. These should be done in a way that makes concurrency easier to reason about.
- Two examples of high level concurrency concepts: (1) asynchronous method calls and (2) Active Objects.

  An asynchronous method call is one that doesn't block; caller is notified when method completes. Futures also fall into this category.

  Active objects are objects that exist in their own thread. The active object acts as a monitor – only one method executes at any given time. The active object can make this guarantee without resorting to locks.
- We also need more tool support for concurrency. For example, trying to debug a data race by setting a breakpoint in a debugger usually doesn't work.

  Systematic defect detection methods (i.e., model checking) that explore every possible execution path are effective. However, these methods can only scale to a small number of threads.

  Tools for finding lock contention would also be valuable.

3.5 Lecture – 11/13/2007

3.5.1 Simple Banking Program (hw22)

The Bank Server creates an instance of `ServerSocket`. The server then calls `serverSocket.accept()`. `accept()` blocks until a client connection is made. When a client connects, `accept()` returns a regular `Socket`, which the server can use for bi-directional communications with the client.

If `s` is an instance of `Socket`, then

- `s.getInputStream()` gives you a handle that can be used to read data from the client, and
- `s.getOutputStream()` gives you a handle that can be used to write data to the client.

In the bank server, `executeCommand` plays the role of a connection handler.

In hw22, we'll implement a new version of the banking program where the server handles each connection in a dedicated thread (n concurrent client connections will give n handler threads on the server). We'll do this by implementing a new Runnable object (say `SBPHandler`) that processes client connections. i.e.,

```
/* 's' is a socket */
Thread t = new Thread(new SBPHandler(s))
t.start();
```

Server sockets are used only for accepting incoming connections.

Regular sockets are used to communicate with clients. Each client connection creates a new regular socket.

3.5.2 Possible Extension to the SBP Program

We could make the following extension to the SBP program: on the client side, have the communications take place in a separate thread. This way, the client's main thread can continue to perform work while the client is waiting for a server response.

For example, the client could implement a runnable called `Connector`. `Connector` is where all of the client/server communications take place.

3.5.3 More client/server designs

Suppose we were going to implement a web server and a web browser. We might design something like the following:

- A `HTTPServer` accepts connections from clients.
- A `HTTPHandler` (a Runnable) processes client connections.
- `HTTPHandler` talks to an `AccessCounter` (to keep track of request statistics), and to a `FileCache` to retrieve HTML files.
- The client uses a `Connector` (a Runnable) to execute communications in a separate thread. The client might use several `Connector` objects; for example, to download several images, js, or css files simultaneously.
- The client might use `Future` objects to render images. Initially, the client might show a bounding box based on attributes in the `img` tag, then replace the bounding box with a real image once the real image has loaded.

Figure 3.8 shows a (simple) diagram of object calls in a HTTP client/server.

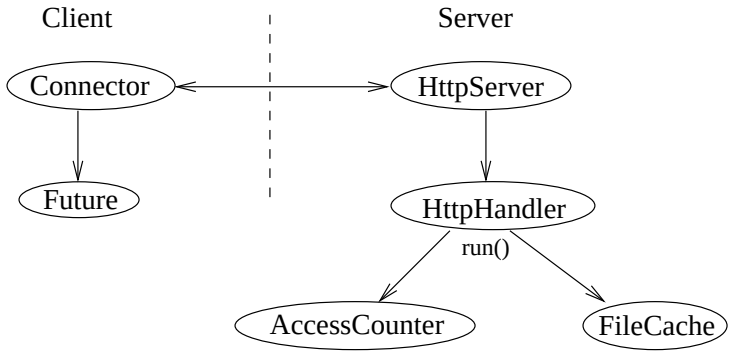

Figure 3.8: Components of a HTTP Client/Server

3.5.4 Concurrency Patterns in Client/Server Applications

Earlier in the semester, we discussed method invocation patterns for multi-threaded applications (see section 2.8.4, page 63). We can apply similar concepts to client/server communications.

Synchronous Communications

The simplest technique is Synchronous Communication. The client sends the request, and blocks until the server response. This is illustrated in Figure 3.9. In Figure 3.9, the client is blocked between points A and B.

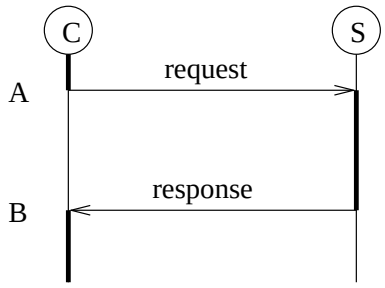

Figure 3.9: Synchronous Client Request

Asynchronous Communications

In asynchronous communications, the client will

- Spawn a separate thread (call it T) to communicate with the server.
- Arrange for the main thread to receive results from T. Either T will call back to the main thread, or the main thread will poll T.

In this scenario, T communicates synchronously with the server, but the client's main thread is free to do other work while the client waits for the server to respond. This is illustrated in Figure 3.10. Note that the client's main thread is free to do work between points A and B.

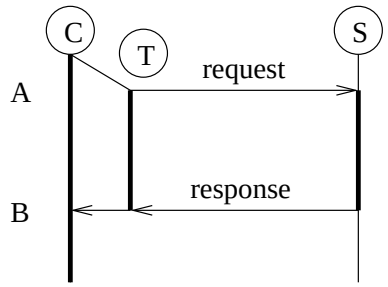

Figure 3.10: Asynchronous Client Request

One-way Communications

In one-way communications, the client sends a request to the server, but does not wait for the server to respond. This is illustrated in Figure 3.11. The creation of a separate thread T (on the client side, to make the request) is optional.

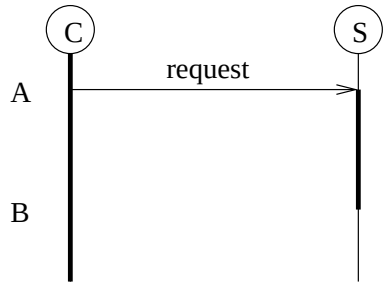

Figure 3.11: One-way Client Request

In the simple banking program, one-way could be used for withdraw or deposit requests; one-way would not be suitable for balance requests.

Reliable One-way Communications

Reliable one-way is a hybrid of the one-way and asynchronous communications patterns. In reliable one-way,

- The client creates a separate thread T that will communicate with the server.
- T makes the request, and waits for the server to respond. T may retry the request if the server does not respond, or if an error occurs.
- T does not pass the server response back to the main thread.

Figure 3.12 illustrates the reliable one-way pattern.

3.5.5 Thread Pools

Suppose your application will handle many concurrent client requests (say, thousands). In this scenario, the tactic of creating a new thread for each client connection may not scale well. Although thread creation is much cheaper than process creation, thread creation is still relatively expensive.

Thread pools provide one way of handling this scenario.

- We create a set of threads when the server starts. These threads are placed in a pool.

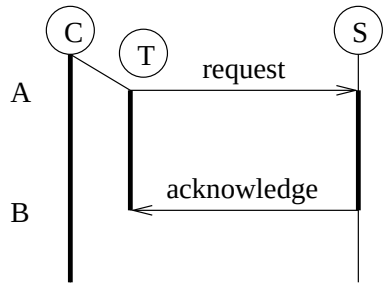

Figure 3.12: Reliable One-way Client Request

- When a client connects, a thread is removed from the pool and assigned the task of handling the client request.
- When the client request finishes, the handler thread is returned to the pool.

Thus, a single thread can handle many different client requests.

Thread pools also give one the option of limiting resource usage. For example, if we set the maximum size of the pool at 500 threads, then no more than 500 client connections would be handled at any single time.

The java standard libraries include a thread pool implementation: `ThreadPoolExecutor`.

3.5.6 Class Logistics

- With the exception of course evaluations, this is our last class meeting for the semester. We should use the rest of the time to work on our projects.
- We will meet on 12/4/2007 for course evaluations.
- Our projects are due on the last day of class: 12/14/2007.

GNU Free Documentation License

Version 1.2, November 2002

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose

the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.