Modern Operating Systems 5th edition Tanenbaum Solutions Manual

SKU: 9780137618941-SOLUTIONS

Simulation Exercises for Operating Systems

Shivakant Mishra

Simulation Exercise 1: Process Management Simulation

Goal: To simulate five process management functions: process creation, replacing the current process image with a new process image, process state transition, process scheduling, and context switching.

You will use Linux system calls such as *fork*( ), *wait*( ), *pipe*( ), and *sleep*( ). Read man pages of these system calls for details.

This simulation exercise consists of three types of Linux processes: *commander*, *process manager*, and *reporter*. There is one commander process (this is the process that starts your simulation), one process manager process that is created by the commander process, and a number of reporter processes that get created by the process manager, as needed.

Commander Process

The commander process first creates a pipe and then a process manager process. It then repeatedly reads commands (one command per second) from the standard input and passes them to the process manager process via the pipe. There are four types of commands:

1. **Q**: End of one unit of time.
2. **U**: Unblock the first simulated process in blocked queue.
3. **P**: Print the current state of the system.
4. **T**: Print the average turnaround time, and terminate the system.

Command **T** appears exactly once, being the last command.

Simulated Process

Process management simulation manages the execution of *simulated* processes. Each simulated process is comprised of a program that manipulates (sets/updates) the value of a single integer variable. Thus the state of a simulated process at any instant is comprised of the value of its integer variable and the value of its program counter. A simulated process' program consists of a sequence of instructions. There are seven types of instructions as follows:

1. **S** *n*: Set the value of the integer variable to *n*, where *n* is an integer.
2. **A** *n*: Add *n* to the value of the integer variable, where *n* is an integer.
3. **D** *n*: Subtract *n* from the value of the integer variable, where *n* is an integer.

MODERN OPERATING SYSTEMS

FIFTH EDITION

PROBLEM SOLUTIONS

ANDREW S. TANENBAUM

HERBERT BOS

Vrije Universiteit
Amsterdam, The Netherlands

PEARSON EDUCATION

HOBOKEN, NJ 07458

SOLUTIONS TO CHAPTER 1 PROBLEMS

1. An operating system must provide the users with an extended machine, and it must manage the I/O devices and other system resources. To some extent, these are different functions.

2. Multiprogramming is the rapid switching of the CPU between multiple processes in memory. It is commonly used to keep the CPU busy while one or more processes are doing I/O.

3. Obviously, there are a lot of possible answers. Here are some.
Mainframe operating system: Claims processing in an insurance company.
Server operating system: Speech-to-text conversion service for Siri.
Multiprocessor operating system: Video editing and rendering.
Personal computer operating system: Word processing application.
Handheld computer operating system: Context-aware recommendation system.
Embedded operating system: Programming a DVD recorder for recording TV.
Sensor-node operating system: Monitoring temperature in a wilderness area.
Real-time operating system: Air traffic control system.
Smart-card operating system: Electronic payment.

4. Empirical evidence shows that memory access exhibits the principle of locality of reference, where if one location is read then the probability of accessing nearby locations next is very high, particularly the following memory locations. So, by caching an entire cache line, the probability of a cache hit next is increased. Also, modern hardware can do a block transfer of 32 or 64 bytes into a cache line much faster than reading the same data as individual words.

5. Input spooling is the technique of reading in jobs, for example, from cards, onto the disk, so that when the currently executing processes are finished, there will be work waiting for the CPU. Output spooling consists of first copying printable files to disk before printing them, rather than printing directly as the output is generated. Input spooling on a personal computer is not very likely, but output spooling is.

6. The prime reason for multiprogramming is to give the CPU something to do while waiting for I/O to complete. If there is no DMA, the CPU is fully occupied doing I/O, so there is nothing to be gained (at least in terms of CPU utilization) by multiprogramming. No matter how much I/O a program does, the CPU will be 100% busy. This of course assumes the major delay is the wait while data are copied. A CPU could do other work if the I/O were slow for other reasons (arriving on a serial line, for instance).

7. Second-generation computers did not have the necessary hardware to protect the operating system from malicious user programs.

8. Access to I/O devices (e.g., a printer) is typically restricted for different users. Some users may be allowed to print as many pages as they like, some users may not be allowed to print at all, while some users may be limited to printing only a certain number of pages. These restrictions are set by system administrators based on some policies. Such policies need to be enforced so that user-level programs cannot interfere with them.

9. A 25 × 80 character monochrome text screen requires a 2000-byte buffer. The 1024 × 768 pixel 24-bit color bitmap requires 2,359,296 bytes. In 1980,these two options would have cost $10 and $11,520, respectively. For current prices, check on how much RAM currently costs, probably pennies per MB.

10. Consider fairness and real time. Fairness requires that each process be allocated its resources in a fair way, with no process getting more than its fair share. On the other hand, real time requires that resources be allocated based on the times when different processes must complete their execution. A real-time process may get a disproportionate share of the resources.

11. Most modern CPUs provide two modes of execution: kernel mode and user mode. The CPU can execute every instruction in its instruction set and use every feature of the hardware when executing in kernel mode. However, it can execute only a subset of instructions and use only subset of features when executing in the user mode. Having two modes allows designers to run user programs in user mode and thus deny them access to critical instructions.

12. Number of heads = 255 GB / (65536*255*512) = 16
Number of platters = 16/2 = 8
The time for a read operation to complete is seek time + rotational latency + transfer time. The seek time is 11 msec, the rotational latency is 7 msec, and the transfer time is 1 msec, so the average transfer takes 19 msec.

13. It may take 20, 25 or 30 msec to complete the execution of these programs depending on how the operating system schedules them. If *P0* and *P1* are scheduled on the same CPU and *P2* is scheduled on the other CPU, it will take 20 msec. If *P0* and *P2* are scheduled on the same CPU and *P1* is scheduled on the other CPU, it will take 25 msec. If *P1* and *P2* are scheduled on the same CPU and *P0* is scheduled on the other CPU, it will take 30 msec. If all three are on the same CPU, it will take 35 msec.

14. Personal computer systems are always interactive, often with only a single user. Mainframe systems nearly always emphasize batch or timesharing with many users. Protection is much more of an issue on mainframe systems, as is efficient use of all resources.

15. Every nanosecond one instruction emerges from the pipeline. This means the machine is executing 1 billion instructions per second. It does not matter at all how many stages the pipeline has. A 10-stage pipeline with 1 nsec per stage would also execute 1 billion instructions per second. All that matters is how often a finished instruction pops out the end of the pipeline.

16. Average access time =
0.95 × 2 nsec (word is in the cache)
+ 0.05 × 0.99 × 10 nsec (word is in RAM, but not in the cache)
+ 0.05 × 0.01 × 10,000,000 nsec (word on disk only)
= 5002.395 nsec
= 5.002395 μsec

17. Maybe. If the caller gets control back and immediately overwrites the data, when the write finally occurs, the wrong data will be written. However, if the driver first copies the data to a private buffer before returning, then the caller can be allowed to continue immediately. Another possibility is to allow the caller to continue and give it a signal when the buffer may be reused, but this is tricky and error prone.

18. A trap is caused by the program and is synchronous with it. If the program is run again and again, the trap will always occur at exactly the same position in the instruction stream. An interrupt is caused by an external event and its timing is not reproducible.

19. Mounting a file system makes any files already in the mount-point directory inaccessible, so mount points are normally empty. However, a system administrator might want to copy some of the most important files normally located in the mounted directory to the mount point so they could be found in their normal path in an emergency when the mounted device was being repaired.

20. A system call allows a user process to access and execute operating system functions inside the kernel. User programs use system calls to invoke operating system services.

21. Prefixing path names with a drive name or number makes the file system device dependent. If the directory on which a new file system is mounted contained any files or subdirectories, those files or subdirectories would not be accessible while the administrator tries to find the backup tape.

22. Open can fail if the file given does not exist or there is a protection error. Close can fail if the file descriptor does not refer to a valid open file. Lseek can fail if the file descriptor is invalid or the new file position is negative.

23. Time multiplexing: CPU, network card, printer, keyboard.
Space multiplexing: memory, disk.
Both: display.

24. If the call fails, for example because *fd* is incorrect, it can return −1. It can also fail because the disk is full and it is not possible to write the number of bytes requested. On a correct termination, it always returns *nbytes*.

25. It contains the bytes: 8, 2, 8, 1.

26. Time to retrieve the file =
1 * 50 msec (time to move the arm over track 50)
+ 5 msec (time for the first sector to rotate under the head)
+ 10/100 * 1000 msec (read 10 MB)
= 155 msec

27. Block special files consist of numbered blocks, each of which can be read or written independently of all the other ones. It is possible to seek to any block and start reading or writing. This is not possible with character special files.

28. System calls do not really have names, other than in a documentation sense. When the library procedure *read* traps to the kernel, it puts the number of the system call in a register or on the stack. This number is used to index into a table. There is really no name used anywhere. On the other hand, the name of the library procedure is very important, since that is what appears in the program.

29. Yes it can, especially if the kernel is a message-passing system.

30. As far as program logic is concerned, it does not matter whether a call to a library procedure results in a system call. But if performance is an issue, if a task can be accomplished without a system call the program will run faster. Every system call involves overhead time in switching from the user context to the kernel context. Furthermore, on a multiuser system, the operating system may schedule another process to run when a system call completes, further slowing the progress in real time of a calling process.

31. Several UNIX calls have no counterpart in the Win32 API:

Link: a Win32 program cannot refer to a file by an alternative name or see it in more than one directory. Also, attempting to create a link is a convenient way to test for and create a lock on a file.

Mount and umount: a Windows program cannot make assumptions about standard path names because on systems with multiple disk drives the drive-name part of the path may be different.

Chmod: Windows uses access control lists.

Kill: Windows programmers cannot kill a misbehaving program that is not cooperating.

32. Every system architecture has its own set of instructions that it can execute. Thus a Pentium cannot execute SPARC programs and a SPARC cannot execute Pentium programs. Also, different architectures differ in bus architecture used (such as VME, ISA, PCI, MCA, SBus, etc.) as well as the word size of the CPU (usually 32 or 64 bit). Because of these differences in hardware, it is not feasible to build an operating system that is completely portable. A highly portable operating system will consist of two high-level layers—a machine-dependent layer and a machine-independent layer. The machine-dependent layer addresses the specifics of the hardware and must be implemented separately for every architecture. This layer provides a uniform interface on which the machine-independent layer is built. The machine-independent layer has to be implemented only once. To be highly portable, the size of the machine-dependent layer must be kept as small as possible.

33. Separation of policy and mechanism allows OS designers to implement a small number of basic primitives in the kernel. These primitives are simplified, because they are not dependent of any specific policy. They can then be used to implement more complex mechanisms and policies at the user level.

34. The virtualization layer introduces increased memory usage and processor overhead as well as increased performance overhead.

35. The conversions are straightforward:

(a) A micro year is $10^{-6} \times 365 \times 24 \times 3600 = 31.536$ sec.
(b) 1000 m or 1 km.
(c) There are 2^{40} bytes, which is 112,5899,906,842,624 bytes.
(d) It is 6×10^{24} kg.

SOLUTIONS TO CHAPTER 2 PROBLEMS

1. The transition from blocked to running is conceivable. Suppose that a process is blocked on I/O and the I/O finishes. If the CPU is otherwise idle, the process could go directly from blocked to running. The other missing transition, from ready to blocked, is impossible. A ready process cannot do I/O or anything else that might block it. Only a running process can block.

2. You could have a register containing a pointer to the current process-table entry. When I/O completed, the CPU would store the current machine state in the current process-table entry. Then it would go to the interrupt vector for the interrupting device and fetch a pointer to another process-table entry (the service procedure). This process would then be started up.

3. Generally, high-level languages do not allow the kind of access to CPU hardware that is required. For instance, an interrupt handler may be required to enable and disable the interrupt servicing a particular device, or to manipulate data within a process' stack area. Also, interrupt service routines must execute as rapidly as possible.

4. There are several reasons for using a separate stack for the kernel. Two of them are as follows. First, you do not want the operating system to crash because a poorly written user program does not allow for enough stack space. Second, if the kernel leaves stack data in a user program's memory space upon return from a system call, a malicious user might be able to use this data to find out information about other processes.

5. The chance that all four processes are idle is 1/16, so the CPU idle time is 1/16.

6. There is enough room for 14 processes in memory. If a process has an I/O of p, then the probability that they are all waiting for I/O is p^{14}. By equating this to 0.01, we get the equation $p^{14} = 0.01$. Solving this, we get $p = 0.72$, so we can tolerate processes with up to 72% I/O wait.

7. If each job has 50% I/O wait, then it will take 20 minutes to complete in the absence of competition. If run sequentially, the second one will finish 40 minutes after the first one starts. With two jobs, the approximate CPU utilization is $1 - 0.5^2$. Thus, each one gets 0.375 CPU minute per minute of real time. To accumulate 10 minutes of CPU time, a job must run for 10/0.375 minutes, or about 26.67 minutes. Thus, running sequentially the jobs finish after 40 minutes, but running in parallel they finish after 26.67 minutes.

8. The probability that all processes are waiting for I/O is 0.4^5 which is 0.01024 Therefore, CPU utilization = $1 - 0.01024 = 0{:}\,98976$.

9. A Web page might have multiple images and other pieces of embedded content each of which is served from a different Web server. In this case, the browser can spawn separate threads and let each thread obtain data from the server serving contents for this page. This can help reduce page downloading time.

10. The client process can create separate threads; each thread can fetch a different part of the file from one of the mirror servers. This can help reduce downtime. Of course, there is a single network link being shared by all threads. This link can become a bottleneck as the number of threads becomes very large.

11. It would be difficult, if not impossible, to keep the file system consistent. Suppose that a client process sends a request to server process 1 to update a file. This process updates the cache entry in its memory. Shortly thereafter, another client process sends a request to server 2 to read that file. Unfortunately, if the file is also cached there, server 2, in its innocence, will return obsolete data. If the first process writes the file through to the disk after caching it, and server 2 checks the disk on every read to see if its cached copy is up-to-date, the system can be made to work, but it is precisely all these disk accesses that the caching system is trying to avoid.

12. A worker thread will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.

13. Yes. If the server is entirely CPU bound, there is no need to have multiple threads. It just adds unnecessary complexity. As an example, consider a telephone directory assistance number (like 555-1212) for an area with 1 million people. If each (name, telephone number) record is, say, 64 characters, the entire database takes 64 megabytes and can easily be kept in the server's memory to provide fast lookup.

14. When a thread is stopped, it has values in the registers. They must be saved, just as when the process is stopped. the registers must be saved. Multiprogramming threads is no different than multiprogramming processes, so each thread needs its own register save area.

15. Threads in a process cooperate. They are not hostile to one another. If yielding is needed for the good of the application, then a thread will yield. After all, it is usually the same programmer who writes the code for all of them.

16. In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is $2/3 \times 15 + 1/3 \times 90$. Thus, the mean request takes 40 msec and the server can do 25 per second. For a multithreaded server, all the waiting for the disk is overlapped, so every request takes 15 msec, and the server can handle 66 2/3 requests per second.

17. The biggest advantage is the efficiency. No traps to the kernel are needed to switch threads. The biggest disadvantage is that if one thread blocks, the entire process blocks.

18. Yes, it can be done. After each call to *pthread_create*, the main program could do a *pthread_join* to wait until the thread just created has exited before creating the next thread.

19. It is possible that one thread will execute

```
accounts[account_count] = entered_account;
```

and will then get blocked. Then another thread will execute the same line of code, modifying the value at *accounts[account_count], since accounts* is a shared resource.

20. The pointers are really necessary because the size of the global variable is unknown. It could be anything from a character to an array of floating-point numbers. If the value were stored, one would have to give the size to *create_global*, which is all right, but what type should the second parameter of *set_global* be, and what type should the value of *read_global* be?

21. It could happen that the runtime system is precisely at the point of blocking or unblocking a thread, and is busy manipulating the scheduling queues. This would be a very inopportune moment for the clock interrupt handler to begin inspecting those queues to see if it was time to do thread switching, since they might be in an inconsistent state. One solution is to set a flag when the runtime system is entered. The clock handler would see this and set its own flag, then return. When the runtime system finished, it would check the clock flag, see that a clock interrupt occurred, and now run the clock handler.

22. Yes it is possible, but inefficient. A thread wanting to do a system call first sets an alarm timer, then does the call. If the call blocks, the timer returns control to the threads package. Of course, most of the time the call will not block, and the timer has to be cleared. Thus each system call that might block has to be executed as three system calls. If timers go off prematurely, all kinds of problems develop. This is not an attractive way to build a threads package.

23. It certainly works with preemptive scheduling. In fact, it was designed for that case. When scheduling is nonpreemptive, it might fail. Consider the case in which *turn* is initially 0 but process 1 runs first. It will just loop forever and never release the CPU.

24. The priority inversion problem occurs when a low-priority process is in its critical region and suddenly a high-priority process becomes ready and is scheduled. If it uses busy waiting, it will run forever. With user-level threads, it cannot happen that a low-priority thread is suddenly preempted to allow a high-priority thread run. There is no preemption. With kernel-level threads this problem can arise.

25. With round-robin scheduling it works. Sooner or later *L* will run, and eventually it will leave its critical region. The point is, with priority scheduling, *L* never gets to run at all; with round robin, it gets a normal time slice periodically, so it has the chance to leave its critical region.

26. Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.

27. A race condition is a situation in which two (or more) processes are about to perform some action. Depending on the exact timing, one or the other goes first. If one of the processes goes first, everything works, but if another one goes first, a fatal error occurs.

28. Yes. The simulated computer could be multiprogrammed. For example, while process *A* is running, it reads out some shared variable. Then a simulated clock tick happens and process *B* runs. It also reads out the same variable. Then it adds 1 to the variable. When process *A* runs, if it also adds 1 to the variable, we have a race condition.

29. Yes, it will work as is. At a given time instant, only one producer (consumer) can add (remove) an item to (from) the buffer.

30. The solution satisfies mutual exclusion since it is not possible for both processes to be in their critical section. That is, when turn is 0, *P0* can execute its critical section, but not *P1*. Likewise, when turn is 1. However, this assumes *P0* must run first. If *P1* produces something and it puts it in a buffer, then while *P0* can get into its critical section, it will find the buffer empty and block. Also, this solution requires strict alternation of the two processes, which is undesirable.

31. Associated with each counting semaphore are two binary semaphores, *M*, used for mutual exclusion, and *B*, used for blocking. Also associated with each counting semaphore is a counter that holds the number of ups minus the number of downs, and a list of processes blocked on that semaphore. To implement down, a process first gains exclusive access to the semaphores, counter, and list by doing a down on *M*. It then decrements the counter. If it is zero or more, it just does an up on *M* and exits. If *M* is negative, the process is put on the list of blocked processes. Then an up is done on *M* and a down is done on *B* to block the process. To implement up, first *M* is downed to get mutual exclusion, and then the counter is incremented. If it is more than zero, no one was blocked, so all that needs to be done is to up *M*. If, however, the counter is now negative or zero, some process must be removed from the list. Finally, an up is done on *B* and *M* in that order.

32. If the program operates in phases and neither process may enter the next phase until both are finished with the current phase, it makes perfect sense to use a barrier.

33. With kernel threads, a thread can block on a semaphore and the kernel can run some other thread in the same process. Consequently, there is no problem using semaphores. With user-level threads, when one thread blocks on a semaphore, the kernel thinks the entire process is blocked and does not run it ever again. Consequently, the process fails.

34. It does not lead to race conditions (nothing is ever lost), but it is effectively busy waiting.

35. It will take nT sec.

36. Three processes are created. After the initial process forks, there are two processes running, a parent and a child. Each of them then forks, creating two additional processes. Then all the processes exit.

37. If a process occurs multiple times in the list, it will get multiple quanta per cycle. This approach could be used to give more important processes a larger share of the CPU. But when the process blocks, all entries had better be removed from the list of runnable processes.

38. In simple cases, it may be possible to see if I/O will be limiting by looking at source code. For instance, a program that reads all its input files into buffers at the start will probably not be I/O bound, but a problem that reads and writes incrementally to a number of different files (such as a compiler) is likely to be I/O bound. If the operating system provides a facility such as the UNIX *ps* command that can tell you the amount of CPU time used by a program, you can compare this with the total time to complete execution of the program. This is, of course, most meaningful on a system where you are the only user.

39. For multiple processes in a pipeline, the common parent could pass to the operating system information about the flow of data. With this information the OS could, for instance, determine which process could supply output to a process blocking on a call for input.

40. If the context switching time is large, then the time quantum value has to be proportionally large. Otherwise, the overhead of context switching can be quite high. Choosing large time quantum values can lead to an inefficient system if the typical CPU burst times are less than the time quantum. If context switching is very small or negligible, then the time quantum value can be chosen with more freedom.

41. The CPU efficiency is the useful CPU time divided by the total CPU time. When $Q \geq T$, the basic cycle is for the process to run for T and undergo a process switch for S. Thus, (a) and (b) have an efficiency of $T/(S + T)$. When the quantum is shorter than T, each run of T will require T/Q process switches, wasting a time ST/Q. The efficiency here is then

$$\frac{T}{T + ST/Q}$$

which reduces to $Q/(Q + S)$, which is the answer to (c). For (d), we just substitute Q for S and find that the efficiency is 50%. Finally, for (e), as $Q \rightarrow 0$ the efficiency goes to 0.

42. Shortest job first is the way to minimize average response time.

$0 < X \leq 3$: $X, 3, 5, 6, 9$.
$3 < X \leq 5$: $3, X, 5, 6, 9$.
$5 < X \leq 6$: $3, 5, X, 6, 9$.
$6 < X \leq 9$: $3, 5, 6, X, 9$.
$X > 9$: $3, 5, 6, 9, X$.

43. For round robin, during the first 10 minutes each job gets 1/5 of the CPU. At the end of 10 minutes, C finishes. During the next 8 minutes, each job gets 1/4 of the CPU, after which time D finishes. Then each of the three remaining jobs gets 1/3 of the CPU for 6 minutes, until B finishes, and so on. The finishing times for the five jobs are 10, 18, 24, 28, and 30, for an average of 22 minutes. For priority scheduling, B is run first. After 6 minutes it is finished. The other jobs finish at 14, 24, 26, and 30, for an average of 18.8 minutes. If the jobs run in the order A through E, they finish at 10, 16, 18, 22, and 30, for an average of 19.2 minutes. Finally, shortest job first yields finishing times of 2, 6, 12, 20, and 30, for an average of 14 minutes.

44. The first time it gets 1 quantum. On succeeding runs it gets 2, 4, 8, and 15, so it must be swapped in five times.

45. A check could be made to see if the program was expecting input and did anything with it. A program that was not expecting input and did not process it would not get any special priority boost.

46. Each voice call needs 200 samples of 1 msec or 200 msec. Together they use 400 msec of CPU time. The video needs 11 msec 33 1/3 times a second for a total of about 367 msec. The sum is 767 msec per second of real time so the system is schedulable.

47. Another video stream consumes 367 msec of time per second for a total of 1134 msec per second of real time so the system is not schedulable.

48. The sequence of predictions is 40, 30, 35, and now 25.

49. The fraction of the CPU used is 35/50 + 20/100 + 10/200 + x/250. To be schedulable, this must be less than 1. Thus x must be less than 12.5 msec.

50. Two-level scheduling is needed when memory is too small to hold all the ready processes. Some set of them is put into memory, and a choice is made from that set. From time to time, the set of in-core processes is adjusted. This algorithm is easy to implement and reasonably efficient, certainly a lot better than, say, round robin without regard to whether a process was in memory or not.

51. Each voice call runs 200 times/second and uses up 1 msec per burst, so each voice call needs 200 msec per second or 400 msec for the two of them. The video runs 25 times a second and uses up 20 msec each time, for a total of 500 msec per second. Together they consume 900 msec per second, so there is time left over and the system is schedulable.

52. The kernel could schedule processes by any means it wishes, but within each process it runs threads strictly in priority order. By letting the user process set the priority of its own threads, the user controls the policy but the kernel handles the mechanism.

53. Variation 1: readers have priority. No writer may start when a reader is active. When a new reader appears, it may start immediately unless a writer is currently active. When a writer finishes, if readers are waiting, they are all started, regardless of the presence of waiting writers. Variation 2: Writers have priority. No reader may start when a writer is waiting. When the last active process finishes, a writer is started, if there is one; otherwise, all the readers (if any) are started. Variation 3: symmetric version. When a reader is active, new readers may start immediately. When a writer finishes, a new writer has priority, if one is waiting. In other words, once we have started reading, we keep reading until there are no readers left. Similarly, once we have started writing, all pending writers are allowed to run.

54. A possible shell script might be

```
if [ ! –f numbers ]; then echo 0 > numbers; fi
count = 0
while (test $count != 200 )
do
  count=′expr $count + 1′
  n=′tail –1 numbers′
  expr $n + 1 >>numbers
done
```

Run the script twice simultaneously, by starting it once in the background