Steven Tang

CSE 460

2/7/14

Lab 5

1. Message queues

Msgctl – What this message queue function does is it performs control operation specified by cmd on the system’s message queue

Msgget – This message queue function GETS a system message queue, returns identifier associated with the value of the key

Msgrcv – This message queue function RECEIVES and READS a message from the queue associated with the queue identifer

Msgsnd – This message queue function SENDS a message to a system message queue

Msg1.cpp

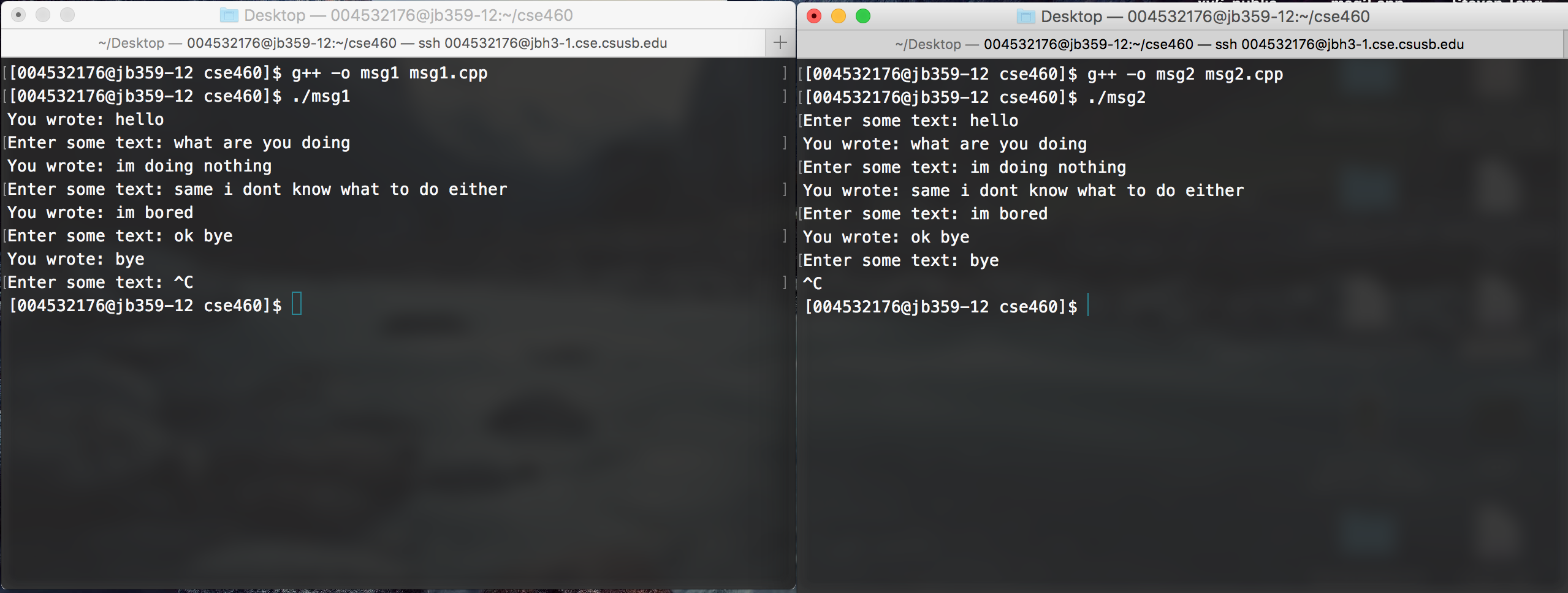
1. //msg1.cpp
2. /\* Here's the receiver program. \*/
4. #include <stdlib.h>
5. #include <stdio.h>
6. #include <string.h>
7. #include <errno.h>
8. #include <unistd.h>
10. #include <sys/types.h>
11. #include <sys/ipc.h>
12. #include <sys/msg.h>

15. **struct** my\_msg\_st {
16. **long** **int** my\_msg\_type;
17. **char** some\_text[BUFSIZ];
18. };
20. **int** main()
21. {
22. **int** running = 1;
23. **int** msgid;
24. **struct** my\_msg\_st some\_data;
25. **long** **int** msg\_to\_receive = 0;
27. /\* First, we set up the message queue. \*/
29. msgid = msgget((key\_t)1234, 0666 | IPC\_CREAT);
31. **if** (msgid == -1) {
32. fprintf(stderr, "msgget failed with error: %d\n", errno);
33. exit(EXIT\_FAILURE);
34. }
35. /\* Then the messages are retrieved from the queue, until an end message is encountered.
36. Lastly, the message queue is deleted. \*/
38. **while**(running) {
39. **if** (msgrcv(msgid, (**void** \*)&some\_data, BUFSIZ,
40. msg\_to\_receive, 0) == -1) {
41. fprintf(stderr, "msgrcv failed with error: %d\n", errno);
42. exit(EXIT\_FAILURE);
43. }
44. printf("You wrote: %s", some\_data.some\_text);
45. **if** (strncmp(some\_data.some\_text, "end", 3) == 0) {
46. running = 0;
47. }
48. }
50. **if** (msgctl(msgid, IPC\_RMID, 0) == -1) {
51. fprintf(stderr, "msgctl(IPC\_RMID) failed\n");
52. exit(EXIT\_FAILURE);
53. }
55. exit(EXIT\_SUCCESS);
56. }

msg2.cpp

1. //msg2.cpp
2. /\* The sender program is very similar to msg1.cpp. In the main set up, delete the
3. msg\_to\_receive declaration and replace it with buffer[BUFSIZ], remove the message
4. queue delete and make the following changes to the running loop.
5. We now have a call to msgsnd to send the entered text to the queue. \*/
7. #include <stdlib.h>
8. #include <stdio.h>
9. #include <string.h>
10. #include <errno.h>
11. #include <unistd.h>
13. #include <sys/types.h>
14. #include <sys/ipc.h>
15. #include <sys/msg.h>
17. #define MAX\_TEXT 512
19. **struct** my\_msg\_st {
20. **long** **int** my\_msg\_type;
21. **char** some\_text[MAX\_TEXT];
22. };
24. **int** main()
25. {
26. **int** running = 1;
27. **struct** my\_msg\_st some\_data;
28. **int** msgid;
29. **char** buffer[BUFSIZ];
31. msgid = msgget((key\_t)1234, 0666 | IPC\_CREAT);
33. **if** (msgid == -1) {
34. fprintf(stderr, "msgget failed with error: %d\n", errno);
35. exit(EXIT\_FAILURE);
36. }
38. **while**(running) {
39. printf("Enter some text: ");
40. fgets(buffer, BUFSIZ, stdin);
41. some\_data.my\_msg\_type = 1;
42. strcpy(some\_data.some\_text, buffer);
44. **if** (msgsnd(msgid, (**void** \*)&some\_data, MAX\_TEXT, 0) == -1) {
45. fprintf(stderr, "msgsnd failed\n");
46. exit(EXIT\_FAILURE);
47. }
48. **if** (strncmp(buffer, "end", 3) == 0) {
49. running = 0;
50. }
51. }
53. exit(EXIT\_SUCCESS);
54. }

message typescript



1. IPCS Status Commands

**IPCS –s**

1. Script started on Tue Feb  7 17:48:35 2017
2. [?1034hbash-3.2$ ipcs -s
3. IPC status from <running system> as of Tue Feb  7 17:48:39 PST 2017
4. T     ID     KEY        MODE       OWNER    GROUP
5. Semaphores:
7. bash-3.2$ exit
8. exit
10. Script done on Tue Feb  7 17:48:44 2017

IPCS –s displays information about any ACTIVE semaphores on the system

**Ipcrm sem** – It asked me to enter a specific id to remove the semaphore, but because there was none running I could not do so.

IPCRM SEM removes the semaphore set from the specific ID that the user has entered.

**Ipcs –m**

1. Script started on Tue Feb  7 17:49:46 2017
2. [?1034hbash-3.2$ ipcs m[K-m
3. IPC status from <running system> as of Tue Feb  7 17:49:52 PST 2017
4. T     ID     KEY        MODE       OWNER    GROUP
5. Shared Memory:
7. bash-3.2$ exit
8. exit
10. Script done on Tue Feb  7 17:49:57 2017

IPCS –M displays information about the active shared memory on the system

**Ipcs –q**

1. Script started on Tue Feb  7 17:50:14 2017
2. [?1034hbash-3.2$ ipcs -[K-q
3. IPC status from <running system> as of Tue Feb  7 17:50:16 PST 2017
4. T     ID     KEY        MODE       OWNER    GROUP
5. Message Queues:
6. q  65536 0x000004d2 --rw-rw-rw- steventang    staff
8. bash-3.2$ ei[Kxit
9. exit
11. Script done on Tue Feb  7 17:50:21 2017

IPCS –Q Displays information about the message queues

1. Study of XV6

Debugging Process:

1. [004532176@jb359-16 xv6]$ gdb
2. GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-94.el7
3. Copyright (C) 2013 Free Software Foundation, Inc.
4. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5. This is free software: you are free to change and redistribute it.
6. There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
7. and "show warranty" **for** details.
8. This GDB was configured as "x86\_64-redhat-linux-gnu".
9. For bug reporting instructions, please see:
10. <http://www.gnu.org/software/gdb/bugs/>.
11. warning: File "/students/csci/004532176/Desktop/xv6/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load:/usr/bin/mono-gdb.py:/usr/lib/golang/src/pkg/runtime/runtime-gdb.py".
12. To enable execution of **this** file add
13. add-auto-load-safe-path /students/csci/004532176/Desktop/xv6/.gdbinit
14. line to your configuration file "/u/cse/004532176/.gdbinit".
15. To completely disable **this** security protection add
16. set auto-load safe-path /
17. line to your configuration file "/u/cse/004532176/.gdbinit".
18. For more information about **this** security protection see the
19. "Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
20. info "(gdb)Auto-loading safe path"
21. (gdb) target remote :27907
22. Remote debugging **using** :27907
23. 0x0000fff0 in ?? ()
24. (gdb) file kernel
25. A program is being debugged already.
26. Are you sure you want to change the file? (y or n) y
27. Reading symbols from /students/csci/004532176/Desktop/xv6/kernel...done.
28. (gdb) **break** swtch
29. Breakpoint 1 at 0x8010456c: file swtch.S, line 10.
30. (gdb) **continue**
31. Continuing.
33. Breakpoint 1, swtch () at swtch.S:10
34. 10    movl 4(%esp), %eax
35. (gdb) step
36. 11    movl 8(%esp), %edx
37. (gdb) step
38. 14    pushl %ebp
39. (gdb) step
40. swtch () at swtch.S:15
41. 15    pushl %ebx
42. (gdb) step
43. swtch () at swtch.S:16
44. 16    pushl %esi
45. (gdb) step
46. swtch () at swtch.S:17
47. 17    pushl %edi
48. (gdb) step
49. swtch () at swtch.S:20
50. 20    movl %esp, (%eax)
51. (gdb) step
52. 21    movl %edx, %esp
53. (gdb) step
54. swtch () at swtch.S:24
55. 24    popl %edi
56. (gdb) step
57. swtch () at swtch.S:25
58. 25    popl %esi
59. (gdb) step
60. swtch () at swtch.S:26
61. 26    popl %ebx
62. (gdb) step
63. swtch () at swtch.S:27
64. 27    popl %ebp
65. (gdb) step
66. swtch () at swtch.S:28
67. 28    ret
68. (gdb) step
69. forkret () at proc.c:351
70. 351 {
71. (gdb) step
72. forkret () at proc.c:354
73. 354   release(&ptable.lock);
74. (gdb) step
75. release (lk=<error reading variable: can't compute CFA **for** **this** frame>,
76. lk@entry=0x80112da0 <ptable>) at spinlock.c:48
77. 48  {
78. (gdb) step
79. 49    **if**(!holding(lk))
80. (gdb) **continue**
81. Continuing.
83. Breakpoint 1, swtch () at swtch.S:10
84. 10    movl 4(%esp), %eax
85. (gdb) clear
86. Deleted breakpoint 1
87. (gdb) **break** exec
88. Breakpoint 2 at 0x801009b0: file exec.c, line 12.
89. (gdb) **continue**
90. Continuing.
91. [New Thread 2]
92. [Switching to Thread 2]
94. Breakpoint 2, exec (
95. path=<error reading variable: can't compute CFA **for** **this** frame>,
96. argv=<error reading variable: can't compute CFA **for** **this** frame>,
97. argv@entry=0x8dfffeb0) at exec.c:12
98. 12  {
99. (gdb) **continue**
100. Continuing.
102. Breakpoint 2, exec (
103. path=<error reading variable: can't compute CFA **for** **this** frame>,
104. argv=<error reading variable: can't compute CFA **for** **this** frame>,
105. argv@entry=0x8dffeeb0) at exec.c:12
106. 12  {
107. (gdb) **break**
108. Note: breakpoint 2 also set at pc 0x801009b0.
109. Breakpoint 3 at 0x801009b0: file exec.c, line 12.
110. (gdb) clear
111. Deleted breakpoints 2 3
112. (gdb) backtrace
113. #0  exec (path=<error reading variable: can't compute CFA for this frame>,
114. argv=<error reading variable: can't compute CFA **for** **this** frame>,
115. argv@entry=0x8dffeeb0) at exec.c:12
116. #1  0x801051e3 in sys\_exec () at sysfile.c:418
117. #2  0x80104749 in syscall () at syscall.c:133
118. #3  0x801056d1 in trap (
119. tf=<error reading variable: can't compute CFA **for** **this** frame>) at trap.c:43
120. #4  0x801054bd in alltraps () at trapasm.S:23
121. #5  0x8dffefb4 in ?? ()
122. Backtrace stopped: previous frame inner to **this** frame (corrupt stack?)
123. (gdb) up
124. #1  0x801051e3 in sys\_exec () at sysfile.c:418
125. 418   **return** exec(path, argv);
126. (gdb) list
127. 413       **break**;
128. 414     }
129. 415     **if**(fetchstr(uarg, &argv[i]) < 0)
130. 416       **return** -1;
131. 417   }
132. 418   **return** exec(path, argv);
133. 419 }
134. 420
135. 421 **int**
136. 422 sys\_pipe(**void**)
137. (gdb)

Scheduler function

1. [004532176@jb359-16 xv6]$ gdb
2. GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-94.el7
3. Copyright (C) 2013 Free Software Foundation, Inc.
4. License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
5. This is free software: you are free to change and redistribute it.
6. There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
7. and "show warranty" **for** details.
8. This GDB was configured as "x86\_64-redhat-linux-gnu".
9. For bug reporting instructions, please see:
10. <http://www.gnu.org/software/gdb/bugs/>.
11. warning: File "/students/csci/004532176/Desktop/xv6/.gdbinit" auto-loading has been declined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load:/usr/bin/mono-gdb.py:/usr/lib/golang/src/pkg/runtime/runtime-gdb.py".
12. To enable execution of **this** file add
13. add-auto-load-safe-path /students/csci/004532176/Desktop/xv6/.gdbinit
14. line to your configuration file "/u/cse/004532176/.gdbinit".
15. To completely disable **this** security protection add
16. set auto-load safe-path /
17. line to your configuration file "/u/cse/004532176/.gdbinit".
18. For more information about **this** security protection see the
19. "Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
20. info "(gdb)Auto-loading safe path"
21. (gdb) target remote:27907
22. Remote debugging **using** :27907
23. 0x0000fff0 in ?? ()
24. (gdb) file kernel
25. A program is being debugged already.
26. Are you sure you want to change the file? (y or n) y
27. Reading symbols from /students/csci/004532176/Desktop/xv6/kernel...done.
28. (gdb) **break** scheduler
29. Breakpoint 1 at 0x80103a20: file proc.c, line 281.
30. (gdb) **continue**
31. Continuing.
32. [New Thread 2]
33. [Switching to Thread 2]
35. Breakpoint 1, scheduler () at proc.c:281
36. 281 {
37. (gdb) list
38. 276 //  - swtch to start running that process
39. 277 //  - eventually that process transfers control
40. 278 //      via swtch back to the scheduler.
41. 279 **void**
42. 280 scheduler(**void**)
43. 281 {
44. 282   **struct** proc \*p;
45. 283
46. 284   **for**(;;){
47. 285     // Enable interrupts on this processor.
48. (gdb) list
49. 286     sti();
50. 287
51. 288     // Loop over process table looking for process to run.
52. 289     acquire(&ptable.lock);
53. 290     **for**(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
54. 291       **if**(p->state != RUNNABLE)
55. 292         **continue**;
56. 293
57. 294       // Switch to chosen process.  It is the process's job
58. 295       // to release ptable.lock and then reacquire it
59. (gdb) list
60. 296       // before jumping back to us.
61. 297       proc = p;
62. 298       switchuvm(p);
63. 299       p->state = RUNNING;
64. 300       swtch(&cpu->scheduler, p->context);
65. 301       switchkvm();
66. 302
67. 303       // Process is done running for now.
68. 304       // It should have changed its p->state before coming back.
69. 305       proc = 0;
70. (gdb) list
71. 306     }
72. 307     release(&ptable.lock);
73. 308
74. 309   }
75. 310 }
76. 311
77. 312 // Enter scheduler.  Must hold only ptable.lock
78. 313 // and have changed proc->state. Saves and restores
79. 314 // intena because intena is a property of this
80. 315 // kernel thread, not this CPU. It should
81. (gdb) list
82. 316 // be proc->intena and proc->ncli, but that would
83. 317 // break in the few places where a lock is held but
84. 318 // there's no process.
85. 319 **void**
86. 320 sched(**void**)
87. 321 {
88. 322   **int** intena;
89. 323
90. 324   **if**(!holding(&ptable.lock))
91. 325     panic("sched ptable.lock");
92. (gdb) list
93. 326   **if**(cpu->ncli != 1)
94. 327     panic("sched locks");
95. 328   **if**(proc->state == RUNNING)
96. 329     panic("sched running");
97. 330   **if**(readeflags()&FL\_IF)
98. 331     panic("sched interruptible");
99. 332   intena = cpu->intena;
100. 333   swtch(&proc->context, cpu->scheduler);
101. 334   cpu->intena = intena;
102. 335 }
103. (gdb) list
104. 336
105. 337 // Give up the CPU for one scheduling round.
106. 338 **void**
107. 339 yield(**void**)
108. 340 {
109. 341   acquire(&ptable.lock);  //DOC: yieldlock
110. 342   proc->state = RUNNABLE;
111. 343   sched();
112. 344   release(&ptable.lock);
113. 345 }
114. (gdb) list
115. 346
116. 347 // A fork child's very first scheduling by scheduler()
117. 348 // will swtch here.  "Return" to user space.
118. 349 **void**
119. 350 forkret(**void**)
120. 351 {
121. 352   **static** **int** first = 1;
122. 353   // Still holding ptable.lock from scheduler.
123. 354   release(&ptable.lock);
124. 355
125. (gdb) q

Evaluation: I did all of the parts that was asked for in the lab. In the first part I explored sending messages through different terminals and then edited my programs so that they would switch back and forth every time they send a message. I have found out that working on OSX will not make the program switch terminals, only on a linux based system. Next I learned how to use the IPCS (Inter-process communication status) and its commands. I also got to XV6 to work and debugged it just like how I was asked to. I followed the steps and did my own debugging of scheduler. At the end I would give myself a 20/20

Score: 20/20