

- 1. Last time
 - 2. The shell, part I
 - 3. File descriptors
 - 4. The shell, part II
 - 5. Processes: the OS's view
 - 6. Threads
 - 7. Intro to Concurrency
-

1. Last time

Stack frames, contd.

Syscalls

process/OS control transfers

Git/lab setup

process birth (fork(1))

shell pipeline demo

2. The shell

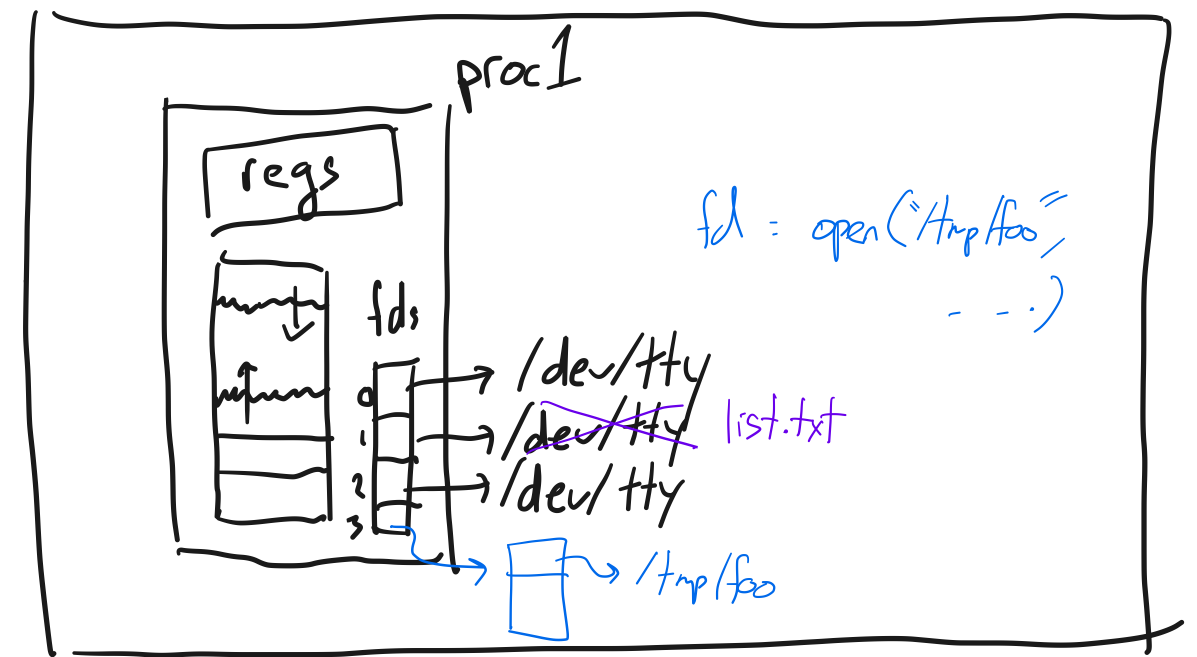
- A program that creates processes
- The human's interface to the computer

Ex:

EX

\$ ls
v

3. File descriptors



printf

write(1, buf, sz)

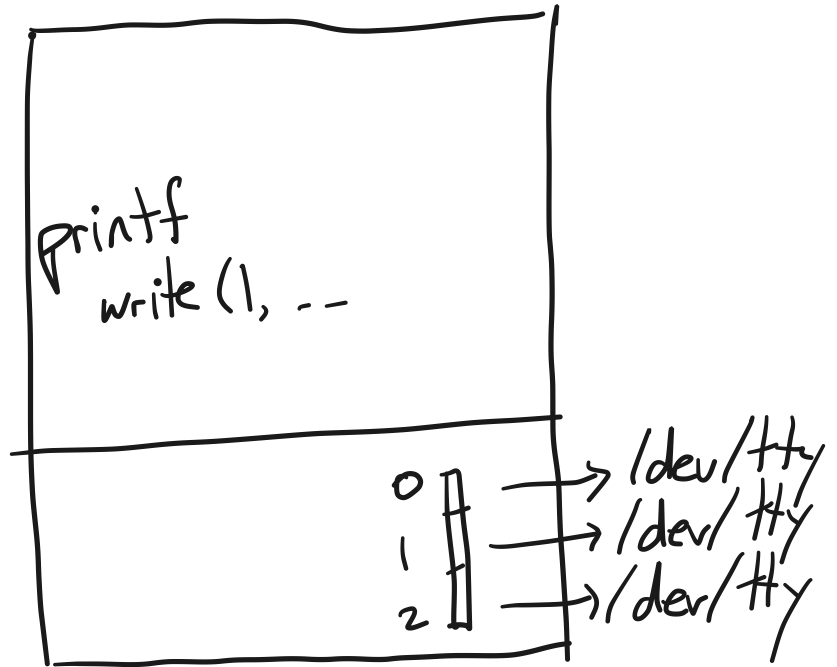
- 0: stdin
- 1: stdout
- 2: stderr

}

4 Shell part II

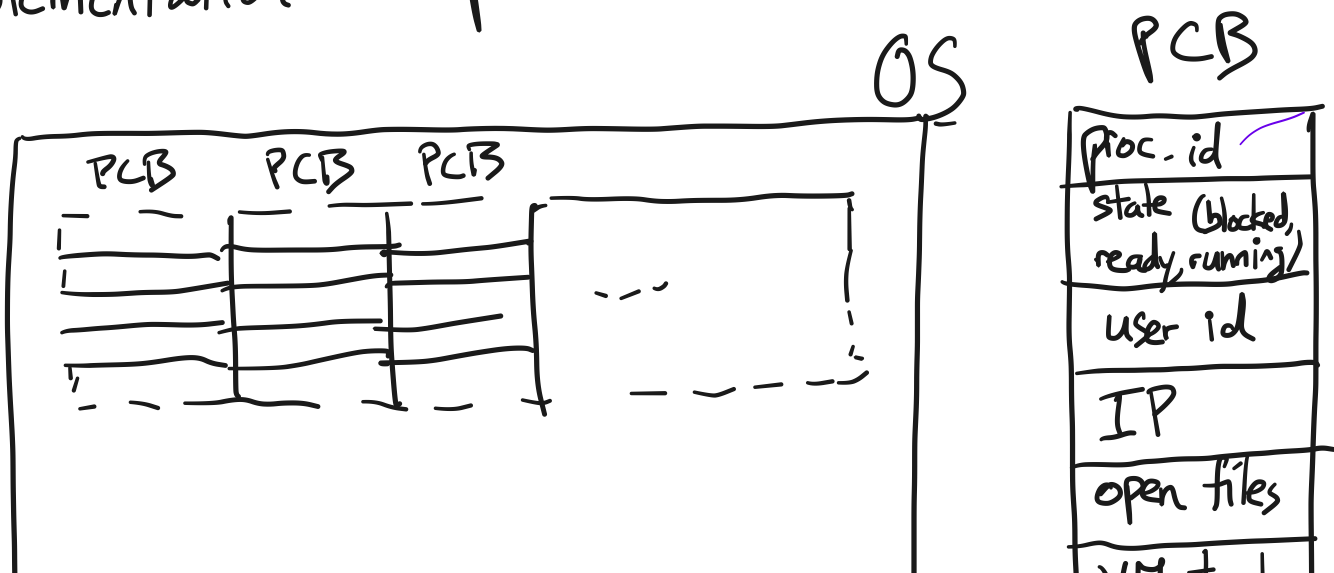
\$./first3 abcd efgh > /tmp/foo

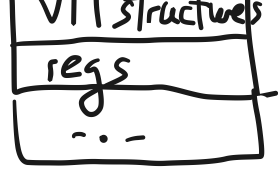
"/tmp/foo"
abc defg



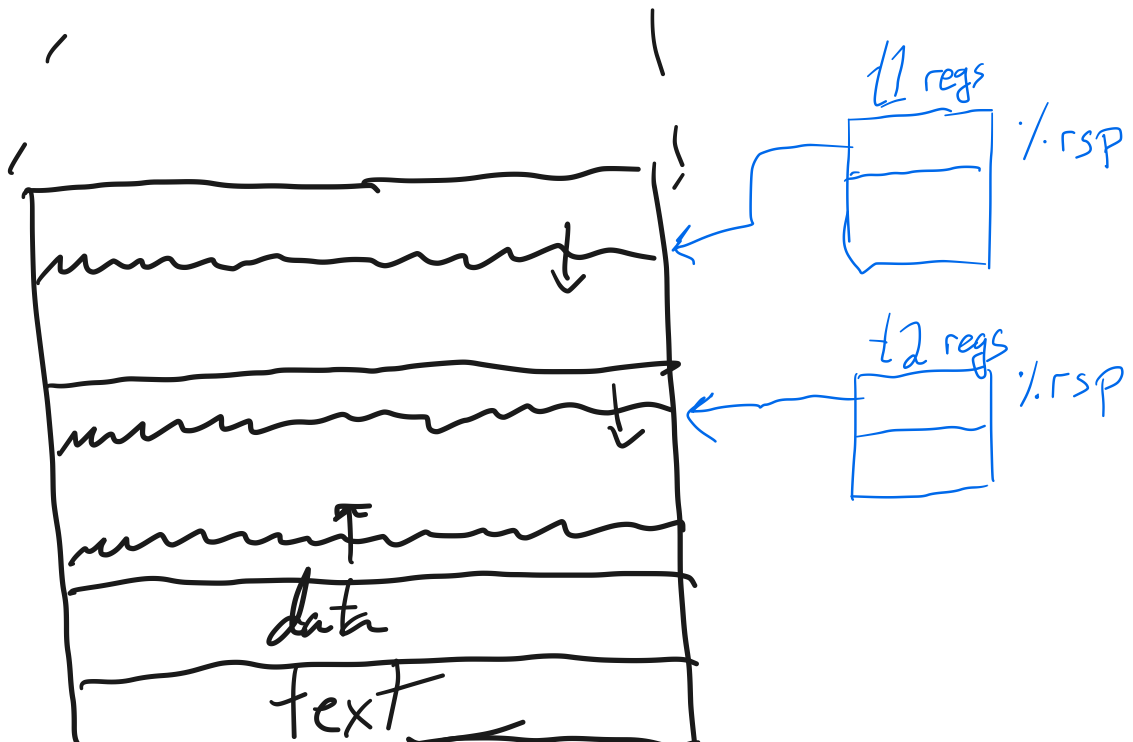
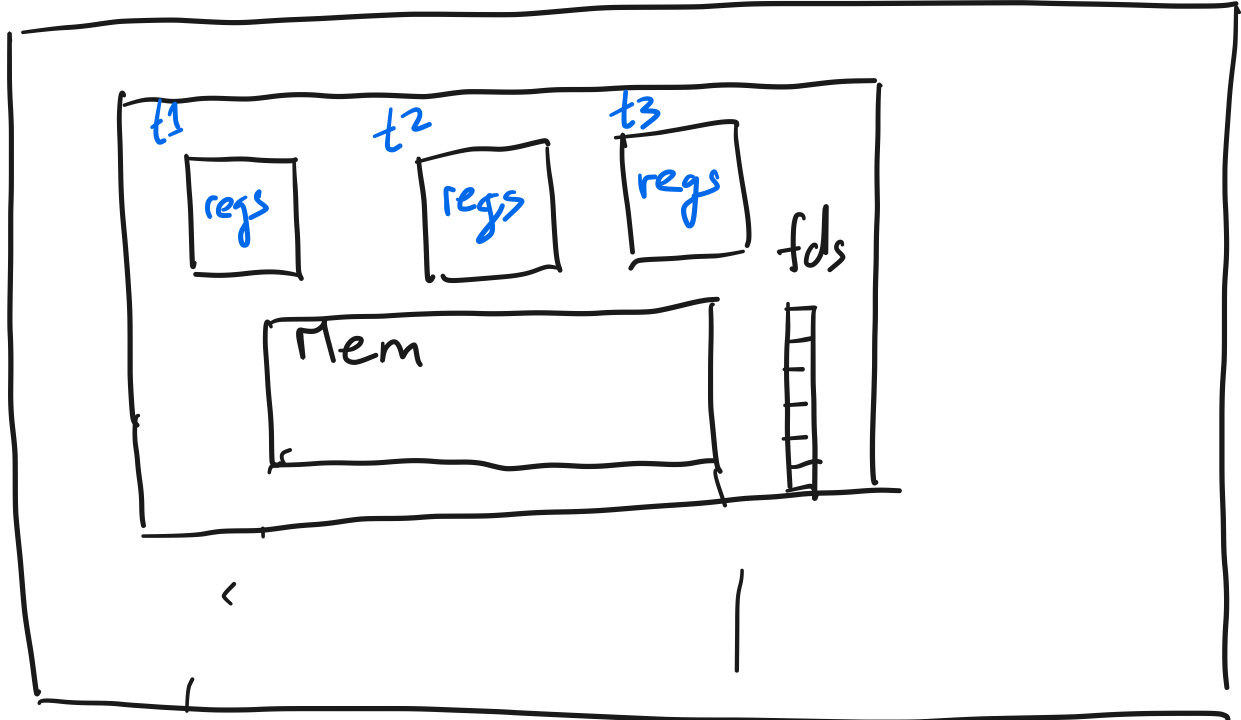
CreateProcess (name, commandline, security-attr,
 ..., ..., ,
 \$: () { : | : & } ; ;

5. Implementation of processes





6. Threads



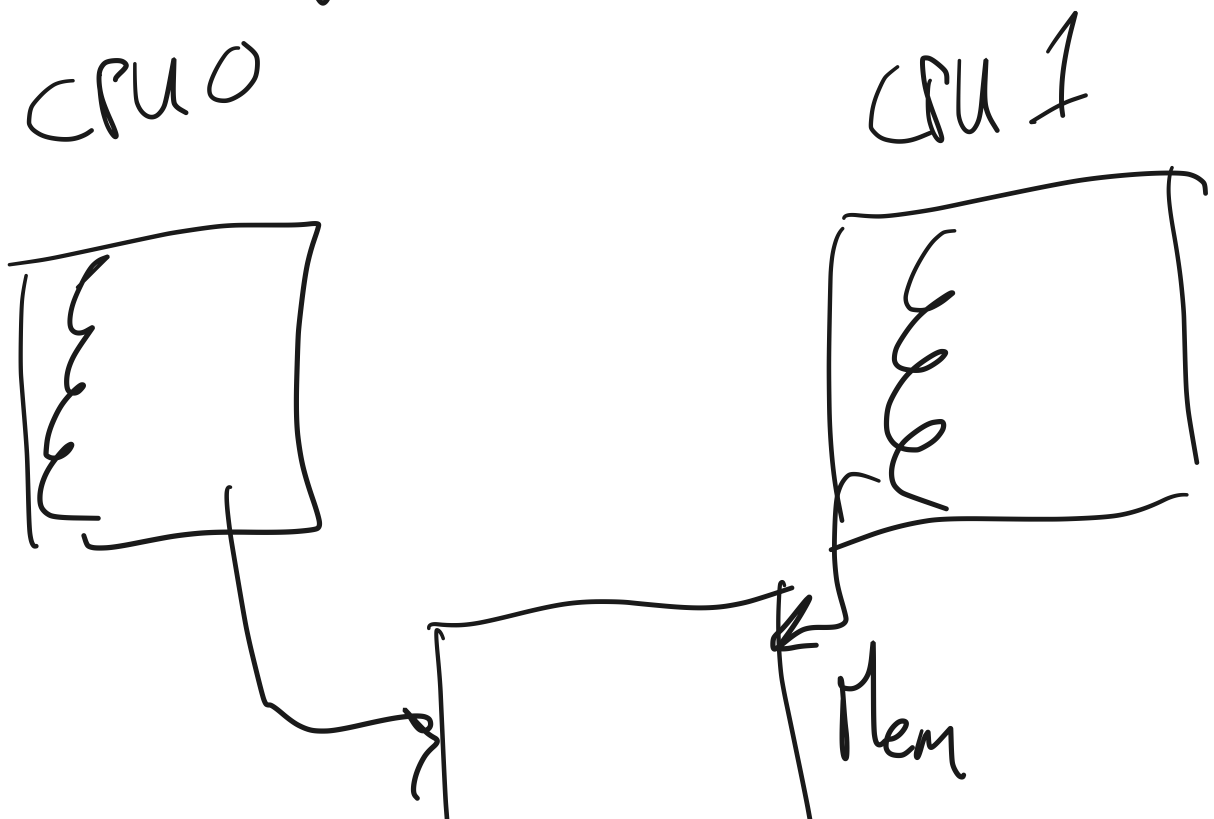
interface

```
tid  thread_create(void (*fn)(void*), void*)  
void  thread_exit();  
void  thread_join(tid thr);
```

+

lots of synchronization primitives

7. concurrency





Feb 01, 23 0:12

handout02.txt

Page 1/4

```
1 CS 202, Spring 2023
2 Handout 2 (Class 4)
```

3
4 The handout is meant to:

- 5 --illustrate how the shell itself uses syscalls
- 6
- 7 --communicate the power of the fork()/exec() separation
- 8
- 9
- 10 --give an example of how small, modular pieces (file descriptors,
- 11 pipes, fork(), exec()) can be combined to achieve complex behavior
- 12 far beyond what any single application designer could or would have
- 13 specified at design time. (We will not cover pipes in lecture today.)

14
15 1. Pseudocode for a very simple shell

```
16 while (1) {
17     write(1, "$ ", 2);
18     readcommand(command, args); // parse input
19     if ((pid = fork()) == 0) // child?
20         execve(command, args, 0);
21     else if (pid > 0) // parent?
22         wait(0); //wait for child
23     else
24         perror("failed to fork");
25 }
26
```

27
28 2. Now add two features to this simple shell: output redirection and
29 backgrounding

30 By output redirection, we mean, for example:

```
31 $ ls > list.txt
```

32 By backgrounding, we mean, for example:

```
33 $ myprog &
34 $
```

```
35
36
37 while (1) {
38     write(1, "$ ", 2);
39     readcommand(command, args); // parse input
40     if ((pid = fork()) == 0) { // child?
41         if (output_redirected) {
42             close(1);
43             open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
44         }
45         // when command runs, fd 1 will refer to the redirected file
46         execve(command, args, 0);
47     } else if (pid > 0) { // parent?
48         if (foreground_process) {
49             wait(0); //wait for child
50         }
51     } else {
52         perror("failed to fork");
53     }
54 }
55
```

Feb 01, 23 0:12

handout02.txt

Page 2/4

56 3. Another syscall example: pipe()

```
57
58 The pipe() syscall is used by the shell to implement pipelines, such as
59 $ ls | sort | head -4
60 We will see this in a moment; for now, here is an example use of
61 pipes.
```

```
62 // C fragment with simple use of pipes
```

```
63
64 int fdarray[2];
65 char buf[512];
66 int n;
67
68 pipe(fdarray);
69 write(fdarray[1], "hello", 5);
70 n = read(fdarray[0], buf, sizeof(buf));
71 // buf[] now contains 'h', 'e', 'l', 'l', 'o'
```

72
73 4. File descriptors are inherited across fork

```
74 // C fragment showing how two processes can communicate over a pipe
```

```
75
76 int fdarray[2];
77 char buf[512];
78 int n, pid;
79
80 pipe(fdarray);
81 pid = fork();
82 if (pid > 0) {
83     write(fdarray[1], "hello", 5);
84 } else {
85     n = read(fdarray[0], buf, sizeof(buf));
86 }
87
88
89
```

Feb 01, 23 0:12

handout02.txt

Page 3/4

```

90 5. Putting it all together: implementing shell pipelines using
91 fork(), exec(), and pipe().
92
93
94 // Pseudocode for a Unix shell that can run processes in the
95 // background, redirect the output of commands, and implement
96 // two element pipelines, such as "ls | sort"
97
98 void main_loop() {
99
100     while (1) {
101         write(1, "$ ", 2);
102         readcommand(command, args); // parse input
103         if ((pid = fork()) == 0) { // child?
104             if (pipeline_requested) {
105                 handle_pipeline(left_command, right_command)
106             } else {
107                 if (output_redirected) {
108                     close(1);
109                     open(redirect_file, O_CREAT | O_TRUNC | O_WRONLY, 0666);
110                 }
111                 exec(command, args, 0);
112             }
113         } else if (pid > 0) { // parent?
114             if (foreground_process) {
115                 wait(0); // wait for child
116             }
117         } else {
118             perror("failed to fork");
119         }
120     }
121 }
122
123 void handle_pipeline(left_command, right_command) {
124
125     int fdarray[2];
126
127     if (pipe(fdarray) < 0) panic ("error");
128     if ((pid = fork ()) == 0) { // child (left end of pipe)
129
130         dup2 (fdarray[1], 1); // make fd 1 the same as fdarray[1],
131                             // which is the write end of the
132                             // pipe. implies close (1).
133
134         close (fdarray[0]);
135         close (fdarray[1]);
136         parse(command1, args1, left_command);
137         exec (command1, args1, 0);
138     } else if (pid > 0) { // parent (right end of pipe)
139
140         dup2 (fdarray[0], 0); // make fd 0 the same as fdarray[0],
141                             // which is the read end of the pipe.
142                             // implies close (0).
143
144         close (fdarray[0]);
145         close (fdarray[1]);
146         parse(command2, args2, right_command);
147         exec (command2, args2, 0);
148     } else {
149         printf ("Unable to fork\n");
150     }
151 }
152

```

Feb 01, 23 0:12

handout02.txt

Page 4/4

```

152
153 6. Commentary
154
155 Why is this interesting? Because pipelines and output redirection
156 are accomplished by manipulating the child's environment, not by
157 asking a program author to implement a complex set of behaviors.
158 That is, the *identical code* for "ls" can result in printing to the
159 screen ("ls -l"), writing to a file ("ls -l > output.txt"), or
160 getting ls's output formatted by a sorting program ("ls -l | sort").
161
162 This concept is powerful indeed. Consider what would be needed if it
163 weren't for redirection: the author of ls would have had to
164 anticipate every possible output mode and would have had to build in
165 an interface by which the user could specify exactly how the output
166 is treated.
167
168 What makes it work is that the author of ls expressed their
169 code in terms of a file descriptor:
170     write(1, "some output", byte_count);
171 This author does not, and cannot, know what the file descriptor will
172 represent at runtime. Meanwhile, the shell has the opportunity, *in
173 between fork() and exec()*, to arrange to have that file descriptor
174 represent a pipe, a file to write to, the console, etc.

```


Jan 31, 23 23:49

our_head.c

Page 1/1

```

1  /*
2  * our_head.c -- a C program that prints the first L lines of its input,
3  *   where L defaults to 10 but can be specified by the caller of the
4  *   program.
5  *
6  *   (This program is inefficient and does not check its error
7  *   conditions. It is meant to illustrate filters aka pipelines.)
8  */
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <stdio.h>
12
13 int main(int argc, char** argv)
14 {
15     int i = 0;
16     int nlines;
17     char ch;
18     int ret;
19
20     if (argc == 2) {
21         nlines = atoi(argv[1]);
22     } else if (argc == 1) {
23         nlines = 10;
24     } else {
25         fprintf(stderr, "usage: our_head [nlines]\n");
26         exit(1);
27     }
28
29     for (i = 0; i < nlines; i++) {
30
31         do {
32
33             /* read in the first character from fd 0 */
34             ret = read(0, &ch, 1);
35
36             /* if there are no more characters to read, then exit */
37             if (ret == 0) exit(0);
38
39             write(1, &ch, 1);
40
41         } while (ch != '\n');
42
43     }
44
45     exit(0);
46 }

```

Jan 31, 23 23:49

our_yes.c

Page 1/1

```

1  /*
2  * our_yes.c -- a C program that prints its argument to the screen on a
3  *   new line every second.
4  *
5  */
6  #include <stdlib.h>
7  #include <string.h>
8  #include <unistd.h>
9  #include <stdio.h>
10
11 int main(int argc, char** argv)
12 {
13     char* repeated;
14     int len;
15
16     /* check to make sure the user gave us one argument */
17     if (argc != 2) {
18         fprintf(stderr, "usage: our_yes string_to_repeat\n");
19         exit(1);
20     }
21
22     repeated = argv[1];
23
24     len = strlen(repeated);
25
26     /* loop forever */
27     while (1) {
28
29         write(1, repeated, len);
30
31         write(1, "\n", 1);
32
33         sleep(1);
34     }
35
36 }

```

1 CS 202, Spring 2023
 2 Handout 3 (Class 4)
 3
 4 1. Example to illustrate interleavings: say that thread tid1 executes f()
 5 and thread tid2 executes g(). (Here, we are using the term "thread"
 6 abstractly. This example applies to any of the approaches that fall
 7 under the word "thread".)

8 a. [this is pseudocode]

```

9     int x;
10
11     int main(int argc, char** argv) {
12
13         tid tid1 = thread_create(f, NULL);
14         tid tid2 = thread_create(g, NULL);
15
16         thread_join(tid1);
17         thread_join(tid2);
18
19         printf("%d\n", x);
20     }
21
22     void f()
23     {
24         x = 1;
25         thread_exit();
26     }
27
28     void g()
29     {
30         x = 2;
31         thread_exit();
32     }
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
    
```

tid1 *tid2*

What are possible values of x after *tid1* has executed f() and *tid2* has executed g()? In other words, what are possible outputs of the program above?

b. Same question as above, but f() and g() are now defined as follows:

```

46     int y = 12;
47
48     f() { x = y + 1; }
49     g() { y = y * 2; }
50
51
52
53
54
55
56
57
58
59
60
61
62
63
    
```

What are the possible values of x?

c. Same question as above, but f() and g() are now defined as follows:

```

58     int x = 0;
59     f() { x = x + 1; }
60     g() { x = x + 2; }
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
    
```

What are the possible values of x?

- ① movq 0x5000, %rbx
- ② addq \$1, %rbx
- ③ movq %rbx, 0x5000
- ④ movq 0x5000, %rbx
- ⑤ addq \$2, %rbx
- ⑥ movq %rbx, 0x5000

64 2. Linked list example

```

65
66     struct List_elem {
67         int data;
68         struct List_elem* next;
69     };
70
71     List_elem* head = 0;
72
73     insert(int data) {
74         List_elem* l = new List_elem;
75         l->data = data;
76         l->next = head;
77         head = l;
78     }
79
80
81
82
83
84
85
86
87
    
```

What happens if two threads execute insert() at once and we get the following interleaving?

thread 1: l->next = head
 thread 2: l->next = head
 thread 2: head = l;
 thread 1: head = l;

Feb 01, 23 0:13

handout03.txt

Page 3/4

```

88 3. Producer/consumer example:
89
90  /*
91  "buffer" stores BUFFER_SIZE items
92  "count" is number of used slots. a variable that lives in memory
93  "out" is next empty buffer slot to fill (if any)
94  "in" is oldest filled slot to consume (if any)
95  */
96
97  void producer (void *ignored) {
98      for (;;) {
99          /* next line produces an item and puts it in nextProduced */
100         nextProduced = means_of_production();
101         while (count == BUFFER_SIZE)
102             ; // do nothing
103         buffer [in] = nextProduced;
104         in = (in + 1) % BUFFER_SIZE;
105         count++;
106     }
107 }
108
109 void consumer (void *ignored) {
110     for (;;) {
111         while (count == 0)
112             ; // do nothing
113         nextConsumed = buffer[out];
114         out = (out + 1) % BUFFER_SIZE;
115         count--;
116         /* next line abstractly consumes the item */
117         consume_item(nextConsumed);
118     }
119 }
120
121 /*
122 what count++ probably compiles to:
123 reg1 <-- count      # load
124 reg1 <-- reg1 + 1   # increment register
125 count <-- reg1     # store
126
127 what count-- could compile to:
128 reg2 <-- count     # load
129 reg2 <-- reg2 - 1  # decrement register
130 count <-- reg2    # store
131 */
132
133 What happens if we get the following interleaving?
134
135 reg1 <-- count
136 reg1 <-- reg1 + 1
137 reg2 <-- count
138 reg2 <-- reg2 - 1
139 count <-- reg1
140 count <-- reg2
141

```

Feb 01, 23 0:13

handout03.txt

Page 4/4

```

142
143 4. Some other examples. What is the point of these?
144
145 [From S.V. Adve and K. Gharachorloo, IEEE Computer, December 1996,
146 66-76. http://rsim.cs.uiuc.edu/~sadve/Publications/computer96.pdf]
147
148 a. Can both "critical sections" run?
149
150     int flag1 = 0, flag2 = 0;
151
152     int main () {
153         tid id = thread_create (p1, NULL);
154         p2 (); thread_join (id);
155     }
156
157     void p1 (void *ignored) {
158         flag1 = 1;
159         if (!flag2) {
160             critical_section_1 ();
161         }
162     }
163
164     void p2 (void *ignored) {
165         flag2 = 1;
166         if (!flag1) {
167             critical_section_2 ();
168         }
169     }
170
171 b. Can use() be called with value 0, if p2 and p1 run concurrently?
172
173     int data = 0, ready = 0;
174
175     void p1 () {
176         data = 2000;
177         ready = 1;
178     }
179     int p2 () {
180         while (!ready) {}
181         use(data);
182     }
183
184 c. Can use() be called with value 0?
185
186     int a = 0, b = 0;
187
188     void p1 (void *ignored) { a = 1; }
189
190     void p2 (void *ignored) {
191         if (a == 1)
192             b = 1;
193     }
194
195     void p3 (void *ignored) {
196         if (b == 1)
197             use (a);
198     }
199

```