```cpp
#include <fcntl.h>
#include <netinet/in.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>

#include <atomic>
#include <cerrno>
#include <cstdio>
#include <cstdlib>
#include <vector>

using namespace std;

#define MSG_NOSIGNAL 0

// Driver stuff
#define CSGAMES_MAX_STARTUPS 8
#define CSGAMES_MAX_STARTUP_NAME_SIZE 100 // including NUL terminator
#define CSGAMES_STARTUP_PREDICTED_YEAR_COUNT 16

#define SIGNGEL SIGUSR1

struct startup {
    char name[CSGAMES_MAX_STARTUP_NAME_SIZE];
    unsigned capital_in_dollars;
};

struct startup_net_worth {
    startup startup;
    long worth_by_year[CSGAMES_STARTUP_PREDICTED_YEAR_COUNT];
};

enum csgames_ioctl_request_code {
    CSGAMES_NEW_STARTUP,        // (const struct startup*)
    CSGAMES_SET_STARTUP,        // (const char*)
    CSGAMES_GET_PROCESSING_END, // (timeval*)
    CSGAMES_GET_ANGELS_PICK,    // (char [CSGAMES_MAX_STARTUP_NAME_SIZE])
};

// Socket stuff
const sockaddr_in bind_addr = {
    .sin_family = AF_INET,
    .sin_port = htons(4321),
    .sin_addr = { htonl(INADDR_LOOPBACK) },
};

```

```
50 // Write all numeric values in network byte order!
51 enum csgames_protocol_command_code {
52     BEGIN_STARTUP,              // (uint32_t, char [])
53     BUSINESS_PLAN_DATA,         // (const char [])
54 
55     ACKNOWLEDGED,               // (void)
56     ANGEL_INVESTOR,             // (void)
57     PREDICTION_RESULT,          // (long [CSGAMES_STARTUP_PREDICTED_YEAR_COUNT])
58     ERROR,                      // (const char [])
59 };
60 
61 // Program
62 struct startup_entry {
63     startup startup;
64     timeval ready_at;
65     pid_t pid;
66 };
67 
68 startup_entry all_startups[CSGAMES_MAX_STARTUPS];
69 atomic_int startup_count;
70 int ss;
71 int lock;
72 
73 struct command_header {
74     unsigned short command;
75     unsigned short size;
76 };
77 
78 struct begin_startup_command {
79     command_header header;
80     unsigned capital;
81     char name[100];
82 };
83 
84 struct business_plan_data_command {
85     command_header header;
86     uint8_t data[1];
87 };
88 
89 struct prediction_result {
90     command_header header;
91     long values[CSGAMES_STARTUP_PREDICTED_YEAR_COUNT];
92 };
93 
94 union command_buffer {
95     command_header header;
96     begin_startup_command begin_startup;
97     business_plan_data_command business_plan_data;
98     prediction_result prediction;
```

```
99      uint8_t bytes[0xffff + sizeof(command_header)];
100 };
101
102 struct lock_file {
103     int fd;
104     lock_file(int fd) : fd(fd) { flock(fd, LOCK_EX); }
105     ~lock_file() { flock(fd, LOCK_UN); }
106 };
107
108 int read_exactly(int sock, void* buffer, ssize_t count) {
109     uint8_t* cBuffer = reinterpret_cast<uint8_t*>(buffer);
110     ssize_t readTotal = 0;
111     while (readTotal != count) {
112         ssize_t result = recv(sock, cBuffer + readTotal, count - readTotal, MSG_NOSI
113         if (result < 0 && errno != EINTR) {
114             return errno;
115         }
116         readTotal += result;
117     }
118     return 0;
119 }
120
121 int send_exactly(int sock, const void* buffer, ssize_t count) {
122     const uint8_t* cBuffer = reinterpret_cast<const uint8_t*>(buffer);
123     ssize_t sentTotal = 0;
124     while (sentTotal != count) {
125         ssize_t result = send(sock, cBuffer + sentTotal, count - sentTotal, MSG_NOSI
126         if (result < 0 && errno != EINTR) {
127             return errno;
128         }
129         sentTotal += result;
130     }
131     return 0;
132 }
133
134 bool before(const timeval& a, const timeval& b) {
135     return a.tv_sec < b.tv_sec || (a.tv_sec == b.tv_sec && a.tv_usec < b.tv_usec);
136 }
137
138 int send_error(int client, const char* fmt, ...) {
139     char* message;
140
141     va_list ap;
142     va_start(ap, fmt);
143     vasprintf(&message, fmt, ap);
144     va_end(ap);
145
146     size_t len = strlen(message);
147     command_header header = {
```

```
148         .command = htons(ERROR),
149         .size = htons(sizeof(command_header) + len)
150     };
151 
152     int result = send_exactly(client, &header, sizeof(header));
153     if (result == 0) {
154         result = send_exactly(client, &message, len);
155     }
156     free(message);
157     return result;
158 }
159 
160 void serve_client(int client, startup_entry* entry) {
161     vector<uint8_t> bytes;
162 
163     // read input
164     while (true) {
165         command_buffer buffer;
166         int result = read_exactly(client, buffer.bytes, sizeof buffer.header);
167         if (result != 0) {
168             bytes.clear();
169             break;
170         }
171 
172         buffer.header.command = ntohs(buffer.header.command);
173         buffer.header.size = ntohs(buffer.header.size);
174         if (buffer.header.size != 0) {
175             result = read_exactly(client, buffer.business_plan_data.data, buffer.hea
176             if (result != 0) {
177                 bytes.clear();
178                 break;
179             }
180         }
181 
182         if (buffer.header.command == BUSINESS_PLAN_DATA) {
183             command_header ack = { ACKNOWLEDGED, sizeof (command_header) };
184             result = send_exactly(client, &ack, sizeof ack);
185             if (result != 0) {
186                 bytes.clear();
187                 break;
188             }
189             if (buffer.header.size == 0) {
190                 bytes.clear();
191                 break;
192             }
193             else {
194                 bytes.insert(bytes.end(), buffer.business_plan_data.data, buffer.byt
195             }
196         }
```

```
197         else {
198             result = send_error(client, "expected BUSINESS_PLAN_DATA command (%i), g
199         }
200     }
201
202     // drain buffer
203     timeval now = {};
204     size_t index = 0;
205     while (index != bytes.size()) {
206         gettimeofday(&now, nullptr);
207         if (before(now, entry->ready_at)) {
208             useconds_t sleep_time = entry->ready_at.tv_sec - now.tv_sec;
209             sleep_time *= 1000000;
210             sleep_time += entry->ready_at.tv_usec - now.tv_usec;
211             usleep(sleep_time);
212             continue;
213         }
214
215         lock_file l(lock);
216         int result = ioctl(ss, CSGAMES_SET_STARTUP, entry->startup.name);
217         if (result < 0) {
218             if (errno == EINTR) {
219                 continue;
220             } else {
221                 break;
222             }
223         }
224         result = write(ss, &bytes[index], bytes.size() - index);
225         if (result < 0) {
226             if (errno != EAGAIN) {
227                 break;
228             }
229         } else {
230             index += result;
231         }
232
233         result = ioctl(ss, CSGAMES_GET_PROCESSING_END, &entry->ready_at);
234         if (result < 0) {
235             break;
236         }
237     }
238
239     // read and send back result
240     startup_net_worth output;
241     {
242         lock_file l(lock);
243         int result = ioctl(ss, CSGAMES_SET_STARTUP, entry->startup.name);
244         if (result < 0) {
245             return;
```

```
246         }
247         result = read(ss, &output, sizeof output);
248         if (result < 0) {
249             return;
250         }
251     }
252 
253     prediction_result result;
254     result.header.command = htons(PREDICTION_RESULT);
255     result.header.size = htons(sizeof result);
256     for (int i = 0; i < CSGAMES_STARTUP_PREDICTED_YEAR_COUNT; ++i) {
257         result.values[i] = htonl(output.worth_by_year[i]);
258     }
259     send_exactly(client, &result, sizeof result);
260 }
261 
262 void handle_client(int client) {
263     command_buffer buffer = {};
264     int result;
265 
266     // THING TO TEST:
267     // How do servers react when clients block here? (This implementation does
268     // not do the right thing.)
269     do
270     {
271         result = read_exactly(client, buffer.bytes, sizeof buffer.header);
272         if (result != 0) {
273             fprintf(stderr, "error %i reading command header!\n", result);
274             return;
275         }
276 
277         buffer.header.command = htons(buffer.header.command);
278         buffer.header.size = htons(buffer.header.size);
279         if (buffer.header.command != BEGIN_STARTUP) {
280             send_error(client, "expected BEGIN_STARTUP!");
281             continue;
282         }
283         if (buffer.header.size > sizeof(begin_startup_command)) {
284             // THING TO TEST:
285             // How do servers react to a buffer that is too large?
286             send_error(client, "BEGIN_STARTUP command is too large!");
287             continue;
288         }
289 
290         result = read_exactly(client, buffer.bytes + sizeof buffer.header, buffer.hea
291         if (result != 0) {
292             fprintf(stderr, "error %i reading command body!\n", result);
293             return;
294         }
```

```
295         buffer.begin_startup.capital = htonl(buffer.begin_startup.capital);
296         buffer.bytes[buffer.header.size] = 0; // nullptr-terminate buffer
297         break;
298     } while (true);
299 
300     pid_t pid = -1;
301     sigset_t set;
302     sigprocmask(0, nullptr, &set);
303     sigset_t oldMask = set;
304 
305     sigaddset(&set, SIGCHLD);
306     sigaddset(&set, SIGNGEL);
307     sigprocmask(SIG_SETMASK, &set, nullptr);
308 
309     int index = startup_count;
310     memset(&all_startups[index], 0, sizeof all_startups[index]);
311     all_startups[index].startup.capital_in_dollars = buffer.begin_startup.capital;
312 
313     size_t nameSize = buffer.begin_startup.header.size
314         - sizeof(buffer.begin_startup.header)
315         - sizeof(buffer.begin_startup.capital);
316     strncpy(all_startups[index].startup.name, buffer.begin_startup.name, sizeof(all
317 
318     result = ioctl(ss, CSGAMES_NEW_STARTUP, &all_startups[index].startup);
319     if (result == 0)
320     {
321         pid = fork();
322         if (pid == -1) {
323             perror("fork");
324         }
325         else if (pid != 0) {
326             all_startups[index].pid = pid;
327             ++startup_count;
328         }
329     }
330     sigprocmask(SIG_SETMASK, &oldMask, nullptr);
331 
332     if (pid == 0) {
333         serve_client(client, &all_startups[index]);
334     }
335 }
336 
337 void sigchld(int, siginfo_t* info, void*) {
338     for (int i = 0; i < startup_count; ++i) {
339         if (info->si_pid == all_startups[i].pid) {
340             memmove(
341                 &all_startups[i],
342                 &all_startups[i+1],
343                 sizeof(all_startups[0]) * (CSGAMES_MAX_STARTUPS - i - 1));
```

```
344             --startup_count;
345             break;
346         }
347     }
348 }
349 
350 void signgel(int) {
351     char startup_name[CSGAMES_MAX_STARTUP_NAME_SIZE];
352     int err = ioctl(ss, CSGAMES_GET_ANGELS_PICK, &startup_name);
353     if (err != 0) {
354         static char message[] = "couldn't get angel's pick!\n";
355         write(STDERR_FILENO, message, sizeof(message)-1);
356         exit(1);
357     }
358 
359     for (int i = 0; i < startup_count; ++i) {
360         if (strncmp(startup_name, all_startups[i].startup.name, sizeof(startup_name
361             kill(all_startups[i].pid, SIGQUIT);
362             return;
363         }
364     }
365     static char message[] = "couldn't find startup!\n";
366     write(STDERR_FILENO, message, sizeof(message)-1);
367 }
368 
369 int main() {
370     int result;
371 
372 #pragma mark - Signal Handlers
373     fprintf(stderr, "Setting up signal handlers... ");
374     struct sigaction chldAction;
375     result = sigaction(SIGCHLD, nullptr, &chldAction);
376     if (result != 0) {
377         perror("sigaction(SIGCHLD)");
378         return 1;
379     }
380     chldAction.sa_sigaction = &sigchld;
381     sigaddset(&chldAction.sa_mask, SIGNGEL);
382     sigaddset(&chldAction.sa_mask, SIGCHLD);
383     result = sigaction(SIGCHLD, &chldAction, nullptr);
384     if (result != 0) {
385         perror("sigaction(SIGCHLD)");
386         return 1;
387     }
388 
389     struct sigaction ngelAction;
390     result = sigaction(SIGNGEL, nullptr, &ngelAction);
391     if (result != 0) {
392         perror("sigaction(SIGNGEL)");
```

```
393         return 1;
394     }
395     ngelAction.sa_handler = &signgel;
396     sigaddset(&ngelAction.sa_mask, SIGNGEL);
397     sigaddset(&ngelAction.sa_mask, SIGCHLD);
398     result = sigaction(SIGNGEL, &ngelAction, nullptr);
399     if (result != 0) {
400         perror("sigaction(SIGNGEL)");
401         return 1;
402     }
403     fprintf(stderr, "done\n");
404 
405 #pragma mark - Preparing lock
406     char lockFileName[] = "csgames_startup_simulator.XXXXXX";
407     lock = mkstemp(lockFileName);
408 
409 #pragma mark - Opening sockets
410     int acceptor = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
411     if (acceptor < 0) {
412         perror("socket");
413         return 1;
414     }
415 
416     int yes = 1;
417     result = setsockopt(acceptor, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));
418     if (result != 0) {
419         perror("setsockopt");
420         return 1;
421     }
422 
423     result = bind(acceptor, (const struct sockaddr*)&bind_addr, sizeof(bind_addr));
424     if (result != 0) {
425         perror("bind");
426         return 1;
427     }
428 
429     result = listen(acceptor, 24);
430     if (result != 0) {
431         perror("listen");
432         return 1;
433     }
434 
435     while (1) {
436         // THING TO TEST:
437         // how do servers handle more than 8 clients?
438         while (startup_count == 8) {
439             pause();
440         }
441 
```

```
442         struct sockaddr_in client_address;
443         socklen_t client_address_length = sizeof(client_address);
444         int client = accept(acceptor, (struct sockaddr*)&client_address, &client_ad
445         if (client < 0) {
446             perror("accept");
447             continue;
448         }
449 
450         // THINGS TO TEST:
451         // How does a client handle being stuck waiting?
452         handle_client(client);
453         close(client);
454     }
455 }
456 
```