CS 451/551 Final Solution

Date: May 19, 2016 **Max Points**: 100

Open book, open notes. No electronic devices

120 Minutes

Please justify all your answers.

Important Reminder As per the course Academic Honesty Statement, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

1. The Unix command `yes` repeatedly outputs a line containing the single character `y` on standard output. The Unix command `head` copies upto the first 10 lines of its standard input onto standard output, and then terminates.

   Given the following pipeline

   ```
   $ yes | head
   ```

   give precise reasons why both commands terminate. *10-points*

   The termination works as follows:

   a) `head` terminates after outputting 10 `y`'s read from the pipeline. This leaves the pipe without any readers.

   b) When the `yes` program attempts to write the next `y` to the pipe, it receives a `SIGPIPE` signal. Presumably it is not catching this signal.

   c) The default disposition for `SIGPIPE` is to terminate the process. Hence `yes` is terminated.

2. Assume that you are working for an organization which has been successfully using a single machine client-server but does not have any of the source code. What is known is that the client and server communicate over FIFO's:

   1. A client with specific PID *pid* creates private FIFO's *pid*`.in` and *pid*`.out` in some fixed directory.

   2. The client writes its PID *pid* to a well-known FIFO followed by a newline.

   3. The client opens *pid*`.out` for writing and subsequently opens *pid*`.in` for reading.

   The details of the protocol for interacting between the client and server are not known; i.e., the details of how messages are exchanged between a client and the server are unknown. Note that the client PID is only used by the server to identify the private FIFOs.

Your manager wants you to make the server available over a network. Specifically, a user on a remote machine should be able to interact with the original server in a manner identical to a user using the original client on the original machine.

Discuss how you could make this happen. Details about client termination and clean-up can be omitted. You may assume that any machines on the network which are to serve as client machines can run the same binaries as the original machine. *15-points*

The basic solution is to wrap the server and client so that the original programs still talk via FIFOs, but the wrapped programs communicate via the network.

On the server machine, an additional network daemon would be run which will forward any requests to the original server. Specifically, this network daemon will listen on some well-known port. When a TCP connection is received, it will fork a worker process to handle the accepted connection. This worker process will use a sequence number *seq* to stand-in for a client PID. It will create private FIFOs *seq*'.in `and "seq"`.out' in the fixed directory. It will then write *seq* to the well-known FIFO followed by a newline. Hence this worker process is all-set to simulate an original client process.

Since nothing can be assumed about the interaction patterns of the original client and server, the worker should transparently connect the network to the two FIFOs without blocking on either the network or the *seq*`.in` FIFO. This can be done using any of the methods discussed in class for monitoring multiple descriptors; an easy possibility is to use multiple threads:

- A receiving thread which opens the *seq*`.out` FIFO for writing and writes anything received over the network to it.
- A sending thread which opens FIFO *seq*`.in` for reading and writes anything received from it to the network socket.

On a client machine, a similar network daemon process will be used to connect any instances of the original client program to the server machine. Specifically, this network daemon process will create and monitor the well-known FIFO. If a client PID *pid* is read, then the daemon forks a worker process.

The worker process would make a client TCP connection to the well-known port on the server machine. It too would create 2 threads:

- A receiving thread which opens the *pid*`.in` FIFO for writing and writes anything received over the network to it.
- A sending thread which opens FIFO *pid*`.out` for reading and writes anything received from it to the network socket.

The creation of the worker processes on both daemons would avoid zombies using either the double-fork technique, or having the daemon process do a explicit `wait()` within a `SIGCHLD` handler.

3. A program consists of many processes running as part of a single process group with processes in the group being dynamically created and dying. Describe how you would set things up so that the process group leader can discover how many other processes there are currently in the group. Your answer should describe how things should be setup in both the process which is the group leader and the processes which are not group leaders. Your answer may **not** use **any** of the following facilities: files, anonymous pipes, FIFOs, shared memory, message queues, semaphores. *15-points*

   A solution would be to use signals. Specifically, the group leader could use `kill(0, SIGUSR1)` to send a signal to every process in the group. The signal handler for `SIGUSR1` would look like:

```
int sigUsr1(int signo) {
  if (signo != SIGUSR1) return;
  kill(getpgrp(), SIGUSR2);
}
```

   Note that both `kill()` and `getpgrp()` are async-signal-safe and hence can be called from a signal handler.

   The handler for `SIGUSR2` could look simply as follows:

```
int isCollecting = 0;
volatile int numProcesses = 0;

int sigUsr2(int signo) {
  if (signo != SIGUSR2) {
    /* handle unexpected signal */
  }
  if (isCollecting) {
    numProcesses++;
  }
  else {
    /* handle case where some process responded too late */
  }
}
```

   The code in the group leader which would get the number of processes in the group would look like:

```
int getNumProcessesInGroup() {
  numProcesses = 0;
  isCollecting = 1;
  int sleepTime = SLEEP_TIME;
  while ((sleepTime = sleep(sleepTime)) > 0) ;
  isCollecting = 0;
  return numProcesses;
}
```

   Note the use of `sleep()` in a loop to handle the arrival of signals.

   One weakness of this scheme is using a `SLEEP_TIME` to allow the other processes to respond with no sure-fire way of ensuring that all processes have responded. This is alleviated somewhat by the use of the `isCollecting` flag to at least detect the case of some process responding outside the sleep window.

4. The following is an edited log of a `ps` command (the editing has added a header, removed and reordered several lines):

```
$ ps -eo user,pid,ppid,pgid,tpgid,sid,comm | grep $USER
USER       PID  PPID  PGID TPGID   SID COMMAND
umrigar   3390  3373  3373    -1  3373 sshd
umrigar   3393  3390  3393  3393  3393 bash
umrigar  10250  3393 10250  3393  3393 firefox
umrigar  10319 10250 10250  3393  3393 plugin-containe
umrigar  10322     1 10250  3393  3393 GoogleTalkPlugi
umrigar  10349  3393 10349  3393  3393 cat
umrigar  10350  3393 10349  3393  3393 grep
umrigar  10351  3393 10349  3393  3393 wc
umrigar  10148  3393 10148  3393  3393 emacs
umrigar  10178 10148 10178 10416 10178 bash
umrigar  10416 10178 10416 10416 10178 ps
umrigar  10417 10178 10416 10416 10178 grep
umrigar   8323     1  8323    -1  8323 word-countd
$
```

   Describe the relationships between the different processes in the above log. *15-points*

   A job is identified by a `pgid`. A background job will have it `pgid` different from the `tpgid`. Hence the above contains the following jobs: a daemon `sshd` (3390), a foreground `bash` (3393), a background `firefox` (10250), a background `cat` (10349), a background `emacs` (10148), a background `bash` (10178) and a foreground `ps` (10178), and a deamon `word-countd` (8323).

   It looks like a `ssh` connection to `sshd` daemon 3390 has spawned a shell 3393 in session 3393. That shell is running in the foreground, but has spawned 3 background jobs:

   a) A firefox background job with `pgid` 10250 has `plugin-containe` 10319 as a child. A `GoogleTalkPlugi` 10322 is also running as part of the `firefox` process group but its original parent has terminated and it has been adopted by `init`.

   b) A job with `pgid` 10349 containing a `cat` (10349), `grep` (10350) and `wc` (10351); based on the incrementing process id's it is probable (but not definite) that the job consists of `cat | grep | wc`.

   c) A `emacs` background job with `pgid` 10148. It has a interior shell `10178`. Note that this interior shell has been given a new `SID` `10178`; this corresponds to a pseudo-tty (not covered in course). The `ps` (10416) and `grep` (10417) which produced the listing seem to be the foreground job of this shell.

   Finally, there appears to be a `word-countd` daemon (8323).

5. A parent process forks several worker processes which produce fixed-size items of some arbitrary size which are to be consumed by the parent process. Evaluate the advantages and disadvantages of the different IPC mechanisms discussed in this course for implementing a shared fixed-size queue which can be used by the child producer processes to transfer the produced items to the parent consumer process.

The IPC mechanisms which should be evaluated are anonymous pipes, FIFOs, Posix message queues, shared memory and network IPC. *15-points*

The advantages and disadvantages of the different IPC mechanisms are discussed below:

Anonymous pipes

A single anonymous pipe created by the parent before forking the child processes could be a candidate for implementing the queue. The child processes would write the produced items into the queue while the parent would read them from the queue to consume them. Problems in this scheme:

- If the size of each individual produced item is greater than `PIPE_BUF` bytes, then the bytes constituting a single item in the byte may not be contiguous.
- The limit on the size of the queue is fixed by the system and not by the fixed-size queue mentioned in the specifications.

The implementation is likely to be very straight-forward.

FIFOs

Using a FIFO would be just like using a anonymous pipe but would have the additional disadvantage of cluttering the filesystem with a name. Hence a FIFO does not appear useful unless there is a possibility that the program specifications could change to allow unrelated processes to be producers for the main consumer process.

POSIX Message Queues

A single message queue would be a viable candidate provided both the following conditions are met:

a) The size of a queue item is less than the maximum message size.

b) The max number of elements in the queue is less than the limit `mq_maxmsg` on the maximum number of messages in the message queue.

One disadvantage is cluttering up the program with a non-portable POSIX name.

Shared Memory

A privately mapped shared memory segment with some additional synchronization mechanism (like unnamed semaphores) would be a good candidate. There would be no constraint (other than resource limitations) on the size of each queue item or the maximum queue size. One advantage would be no kernel involvement while reading/writing an item from/to the queue (there may be kernel involvement when synchronizing, especially if there is queue contention).

The implementation (including the synchronization) is quite complex, but quite standard (in that code from a text or the web could easily be adapted).

Network IPC

This seems to be a very poor fit to the problem. It has the serious disadvantage of not inherently preserving the identity of queued items. This alternative is not worth serious consideration unless there is a future possibility that the specifications could change to include a network-distributed solution.

The above lists the advantages and disadvantages of each approach. As with any other implementation decision, the chosen implementation should be hidden under a ADT. If nothing is known about how the specifications may evolve, then if the limits imposed by pipes/message queues are acceptable, then go with an anonymous pipe or message queue (with a preference for anonymous pipes); otherwise (or if individual item sizes are large where the efficiency of shared memory can make a difference), go with shared memory.

6. Write a iterative TCP server. When it receives a connection request from a client, it should send to the client a line containing the local time in ISO-8601 format, and then terminates the connection. An ISO-8601 time must be in the format *YYYY*`-`*MM*`-`*DD*`T`*HH*`:`*MM*`:`*SS* with all components except year containing exactly 2 digits; example `2016-02-03T22:08:27`.

   The server is started with a single command-line argument specifying the port on which it should listen for connection requests on all local network interfaces. If the port is under 1024, then the code must check whether the server was started as root.

   You answer must include all the code for setting up the network connection, but need not include code for setting up a daemon. It also need not show error checking for system calls or the inclusion of any necessary header files. *15-points*

```
/** Main server loop.  For each incomining connection on socket spawn
 *  a new process which reads processes incoming aggregate requests.
 */
static void
  serverLoop(int socket)
{
   while (1) {
     struct sockaddr_in rsin;
     socklen_t rlen = sizeof(rsin);
     int s = accept(socket, (struct sockaddr*)&rsin, &rlen);
     FILE *out = fdopen(s, "w");
     time_t now = time(NULL);
     struct tm *local = localtime(&now);
     fprintf(out, "%4d-%02d-%02dT%02d:%02d:%02d\n",
                  local->tm_year + 1900, local->tm_mon + 1,
                  local->tm_mday, local->tm_hour, local->tm_min,
                  local->tm_sec);
     fflush(out);
     fclose(out);
   } /* while (1) */
}

/** Setup a network server on port and loop on incoming client
 *  requests.
 */
static void
```

```
  doServer(int port)
{
  enum { QLEN = 5 };
  int s;
  struct sockaddr_in sin;
  s = socket(PF_INET, SOCK_STREAM, 0);
  memset((char *)&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;
  sin.sin_port = htons((unsigned short)port);
  bind(s, (struct sockaddr *)&sin, sizeof(sin));
  listen(s, QLEN);
  serverLoop(s);
}

/** Main program: changes DIR to first argument.  Creates server
 *  running on port specified by 2nd argument.
 */
int
  main(int argc, const char *argv[])
{
  if (argc != 2) {
    fatal("usage: %s PORT\n", argv[0]);
  }
  else {
    int port = atoi(argv[1]);
    if (port < 1024 && geteuid() != 0) fatal("bad port %s\n", argv[2]);
    doServer(port);
  }
  return 0;
}
```

The full program (including error checking which is not required) is available in ./programs/time-server/.

7. Discuss the validity of the following statements: *15-points*

   a) A program does not have any security flaws if and only if it does not have any buffer overflows

   b) A server creates a POSIX shared memory segment having a specific POSIX name. If a client wants to attach that entire segment into its virtual memory, then it must know both the POSIX name as well as the size of the shared memory segment.

   c) For proper security, passwords must be encrypted before being stored.

   d) A `write()` call is always synchronous and synchronized.

   e) Since TCP guarantees successful data delivery, a `write()` to a TCP socket can never fail.

   The solutions follow:

a) The statement *if a program does not have any security flaws, then it cannot have any buffer overflows* is **true**. However, the converse statement *if a program does not have any buffer overflows, then it does not have any security flaws* is **false**; it could have some other security flaw not involving buffer overflows like using untrusted input as data when executing other programs. Hence the overall **if-and-only-if** statement is **false**.

b) It is sufficient for the client to merely know the POSIX name as it can always determine the size of the shared memory segment using `fstat()`. Hence the statement is **false**.

c) The word `encrypt` implies a corresponding `decrypt` operation. If a `decrypt` operation is possible, then the plaintext password can be recovered from the stored password. However, such recovery of the original plaintext password is not necessary (and in fact, is discouraged). For proper security, it is sufficient that some deterministic hash of the plaintext password is stored instead. Hence the statement is **false**.

d) A `write()` call is always synchronous in that it does not return until the data is written (usually to kernel buffers). It may or may not be synchronized depending on whether the underlying descriptor has the `O_SYNC` flag set.

e) The fact that TCP guarantees successful delivery means that a user of TCP is alerted when delivery fails. Hence it is entirely possible that a write to a TCP socket can fail.