Server Software

Lecture Notes

jerakrs

February 2018

Abstract

This document is the lecture note of *Server Software*. The course code is *COMSM2001* and the unit director is *Steve Gregory*.

1 Introduction

Server is a process running in a concurrent or distributed system providing a service to client processes. While **Client-server paradigm**

- provides access to service or data available at one place.
- provides access to service that users cannot access directly.
- but efficiency can be problem.
- but other paradigms can be more natural.

1.1 Concurrent, Parallel and Distributed

- Server is usually part of a distributed system.
- Server itself may be parallel, for speed and/or reliability.
- Server is usually concurrent, to handle multiple clients.

1.2 Synchronous vs. Asynchronous

- Synchronous
  - need to wait until it can start.
  - caller blocked until the operation finishes.
- Asynchronous
  - can test if something is ready.
  - concurrency – run "in background"

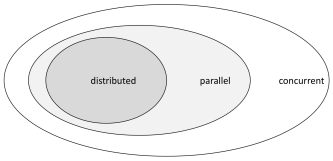


Figure 1: Relationship of concurrent, parallel and distributed

2 C programming

2.1 The Error Checking of File Operation

Listing 1: Example of File Operation

```
1  FILE* file = fopen("name", "r");
2  if (file == NULL) {
3      printf("Error opening file: %s\n", strerror(errno));
4      exit(EXIT_FAILURE);
5  }
6
7  char buffer[80];
8  fscanf(file, "%79s", buffer);
9
10 if (ferror(file)) {
11     printf("Error reading file: %s\n", strerror(errno));
12     exit(EXIT_FAILURE);
13 } else {
14     printf("Hello %s\n", buffer);
15 }
16
17 if (fclose(file) != 0) {
18     printf("Error closing file: %s\n", strerror(errno));
19     exit(EXIT_FAILURE);
20 }
21
22 exit(EXIT_SUCCESS);
```

There are four common problems:

- fail to open file
- buffer overflow
- fail to read/write file
- forget to close file or fail to close file

Library:

- **int errno** (*errno.h*) will set/cleared by C library functions to indicate an error.
- **char* strerror(int errno)** (*string.h*) will return a pointer to a string describing the error code.
- **void exit(int exitno)** (*stdlib.h*) will exit the program, while 0 = success, !0 = error. And exit(i) is the same as "return i" in main().
- **int feof(FILE* file)** (*stdio.h*) will check for end-of file, while $ret = 0$ means file is unfinished, ret =!0 means file is finished.
- **int ferror(FILE* file)** (*stdio.h*) will check for errors, while $ret = 0$ means it has no error, ret =!0 means it has error.

2.2 The Error Checking of Heap Memory

Listing 2: Example of Heap Memory

```
buffer = malloc(BUF_SIZE);
if (buffer == NULL) {
    printf("Error to apply memory.\n");
    exit(EXIT_FAILURE);
}

int err = do_work(buffer);
if (err) {
    printf("Error to use memory.\n");
    exit(EXIT_FAILURE);
}

free(buffer);

exit(EXIT_SUCCESS);
```

There are three common problems:

- fail to apply memory.
- error in using memory.
- forget to free memory.

Library

- **void *malloc(int bytes)** (*stdlib.h*) allocates the specified number of bytes, while $ret = NULL$ means fail, $ret = !NULL$ means success.
- **void free(void *ptr)** (*stdlib.h*) deallocates the memory previously allocated.

3 Processes

Process is the execution of a program. A program can be run as many processes and a program may comprise many processes. Each process has an id, the command *ps [-A]* can show them.

Concurrency: At regular intervals, the processor receives a timer interrupt and switches to the operating system. The process has three states: running, ready, blocked. The OS maintains a list of ready processes – which one gets to run next is determined by a scheduling policy. There is also a list for blocked processes.

3.1 Multiprocess Error Checking

Listing 3: The Example of Multiprocess

```
int pid = fork();

if (pid == 0) {
    printf("I'm the child\n");
    exit(EXIT_SUCCESS);

} else if (pid > 0) {
    printf("I'm the paraent\n");
```

```
9
10      int result;
11      int err = waitpid(pid, &result, 0);
12      if (err == pid) {
13          printf("Child done-returned %i.\n", result);
14      } else {
15          printf("Error.");
16      }
17
18  } else {
19      printf("Fork failed!\n");
20      exit(EXIT_FAILURE);
21  }
22
23  return EXIT_SUCCESS;
```

Library:

- **int fork(void)** (*unistd.h*) will create a clone of the current process, while $ret = -1$ means create child process fail, $ret = 0$ means this is child process and $ret = PID$ means success.
- **int kill(int pid, int sig)** (*signal.h*) will send a signal to a process or a group of processes specified by pid.
  - kill(PID) sends a signal to another process, usually to kill it.
  - kill(-1) sends a signal to all processes.
- **int waitpid(int pid, int *status, int options)** (*sys/types.h*) will wait for child process with identifier pid to terminate and write the return value into status.
  - $pid = -1$ waits for any (one) child.
  - $options = 0$ normally.
  - $option = WNOHANG$ the process will test if there is currently a finished child.

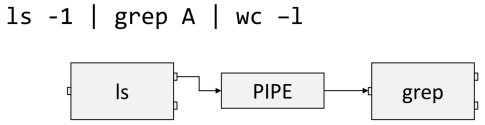
4 I/O

C/POSIX processes interact with "files" through file descriptors. **Everything is a file**: file I/O can be used for pipes, sockets, raw devices

4.1 stdio

The three input/output (I/O) connections are called standard input (stdin 0), standard output (stdout 1) and standard error (stderr 2).

(a) stdio



(b) pipe

4.2 pipes

A pipe is a "special file": its contents do not exist on any disk.

5 Threads

A thread is a "lightweight process".

5.1 Processes Vs Threads

- Processes
  - Multiple simultaneous programs.
  - Independent memory space.
  - Independent open file descriptors.
  - A process is a container for threads.
- Threads
  - Multiple simultaneous functions.
  - Shared memory space.
  - Shared open file descriptors.
  - One copy of the heap.
  - One copy of the code.
  - Multiple stacks.

5.2 Race Conditions

```
...
// THREAD 0
...

  // register interest
  interested[0] = true;

  // secure next available turn
  turn = 0;

  while (
    (interested[1]==true) &&
    (turn == 0)) {
    // busy wait
  }

  // CRITICAL SECTION

  interested[0] = false;
```

```
...
// THREAD 1
...

  // register interest
  interested[1] = true;

  // secure next available turn
  turn = 1;

  while (
    (interested[0]==true) &&
    (turn == 1)) {
    // busy wait
  }

  // CRITICAL SECTION

  interested[1] = false;
```

achieves mutual exclusion

Figure 3: Peterson's Algorithm

5.3 pthread: POSIX threads

thread_creation

```
int pthread_create(
    pthread_t *tidp, const pthread_attr_t *attr,
    (void*)(*start_rtn)(void*), void *arg)
```

- the first argument is a pointer to the pthread_t.

- the third argument is a function, the fourth its argument.
- return an int which is 0 for success and the error code otherwise.

thread_join

```
int pthread_join(pthread_t thread, void **retval);
```

- pthread_join blocks if the thread is still alive.

6 Mutual Exclusion

Mutual Exclusion (mutex) is a property of concurrency control, which is instituted for the purpose of preventing race conditions. The header file is *pthread.h*

Listing 4: mutex

```
1  // no need to destory
2  pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
3
4  // 0: success, !0: fail.
5  pthread_mutex_init(&lock, NULL);
6
7  int pthread_mutex_lock(pthread_mutex_t*);
8  int pthread_mutex_unlock(pthread_mutex_t*);
9
10 // non-blocking function
11 // 0: mutex was free, !0: mutex is locked
12 int pthread_mutex_trylock(pthread_mutex_t*);
13
14 int pthread_mutex_destroy(pthread_mutex_t*);
```

7 Condition Variables

Problem: Two variables x, y, two threads, the first thread will do some operations when $x > y$, the second thread will swap x and y when $x \leq y$.

7.1 Busy Waiting

Listing 5: Thread 1

```
1 while (!done) {
2     LOCK(lock);
3     if (x > y) {
4         OPERATE x, y;
5         done = 1;
6     }
7     UNLOCK(lock);
8 }
```

Listing 6: Thread 2

```
while (!done) {
    LOCK(lock);
    if (x <= y) {
        SWAP x, y;
        done = 1;
    }
    UNLOCK(lock);
}
```

7.2 Condition Variables

When a thread needs to wait for something, it goes to sleep. When a thread does something, it will wakes all threads up. Threads that wake up but can't find anything to do go back to sleep.

Basic operations

- wait – go to sleep until someone wakes you up
- signal – wake up exactly one sleeper
- broadcast – wake up all sleepers

Library

- **pthread_cond_t condition = PTHREAD_COND_INITIALIZER** (*pthread.h*) initialise a static condition variable, which do not need to destroy.
- **int pthread_cond_init(*condition, NULL)** (*pthread.h*) initialises a dynamic condition variable specified by *condition* argument, $ret = 0$ means success, $ret! = 0$ means error.
- **int pthread_cond_destroy(*condition)** (*pthread.h*) destroy a dynamic condition variable specified by *condition* argument, $ret = 0$ means success, $ret! = 0$ means error.
- **int pthread_cond_wait(*condition, *mutex)** (*pthread.h*) thread go to sleep and unlock the mutex; after weak up, it will reapply the metux.
- **int pthread_cond_signal(*condition)** (*pthread.h*) wake exactly one of sleeper.
- **int pthread_cond_broadcast(*condition)** (*pthread.h*) wake all sleepers.

Listing 7: Thread 1

```
1 LOCK(lock);
2 while (x <= y) {
3     WAIT(cond, lock);
4 }
5 OPERATE x, y;
6 UNLOCK(lock);
```

Listing 8: Thread 2

```
LOCK(lock);
while (x > y) {
    WAIT(cond, lock);
}
SWAP x, y;
BROADCAST(cond);
UNLOCK(lock);
```

Benefit: Sleeper will not occupy the CPU resources.

8 Semaphores

Semaphores are an alternative to condition variables. They are not part of the pthreads library and which technique to use depends on the application.

A semaphore has a value ≥ 0. Operations:

- post - increase the value by 1 (atomically).
- wait – if > 0, decrease by 1 (atomically), otherwise wait until this becomes possible.

Library

- **sem_t semaphore** (*semaphore.h*)

- **int sem_init(*sem, pshared, value)** (*semaphore.h*) initialises the unnamed semaphore specified by the *sem* argument, while *pshared* is ignored (usually be zero) and *value* specifies the value to assign to the newly initialised.
- **int sem_destroy(*sem)** (*semaphore.h*) destroy a semaphore specified by the *sem* argument.
- **int sem_wait(*sem)** (*semaphore.h*) locks the specified semaphore by performing a semaphore lock operation on that semaphore, $ret = 0$ means success to lock, $ret = -1$ means error and error message will store in *errno*.
- **int sem_trywait(*sem)** (*semaphore.h*) locks the specified semaphore only if that semaphore is currently not locked.
- **int sem_post(*sem)** (*semaphore.h*) unlocks the specified semaphore by performing a semaphore unlock operation on that semaphore, $ret = 0$ means success to lock, $ret = -1$ means error and error message will store in *errno*.

Listing 9: Semaphore

```
1 sem_wait(&semaphore);
2 printf("Hello from da thread!\n");
3 sem_post(&semaphore);
```

9 Safety and Liveness

9.1 Safety

Program Safety refers to the threads safety and encapsulates issues.

Race Conditions: In multithreaded applications the programmer often expects a particular ordering of instructions, while the certain programming errors can lead to non-deterministic ordering, causing undesired behavior. The program can use the atomic instructions or lockers to avoid the race conditions. Atomic instructions (**Atomicity**) are generally available as compiler extensions and may map to hardware instructions. The lockers allow the programmer to control access to blocks of code, such that we can avoid multiple threads touching the same resources.

Library Functions: The programme should also ensure any functions that the program uses are threads safe.

9.2 Liveness

Guaranteeing **liveness** means that all of the threads are able to make forward progress during their execution. It needs to ensure that threads don't end up waiting on resources that will never be released.

Deadlock occurs because threads will wait indefinitely on locked shared resources. It is a special case of **Forward Progress Bugs**.

10 Communication

Communication Methods:

- Shared memory communication.
- Message passing.
- Asynchronous message passing.

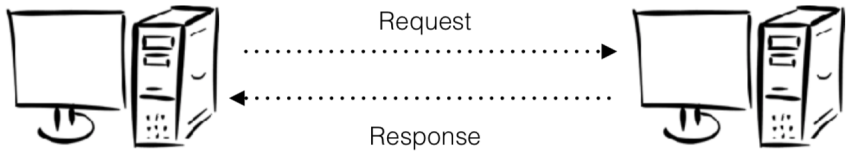


Figure 4: Peer-to-Peer Communication

Distributed systems

- Peer-to-peer communication:
- Server-client communication:

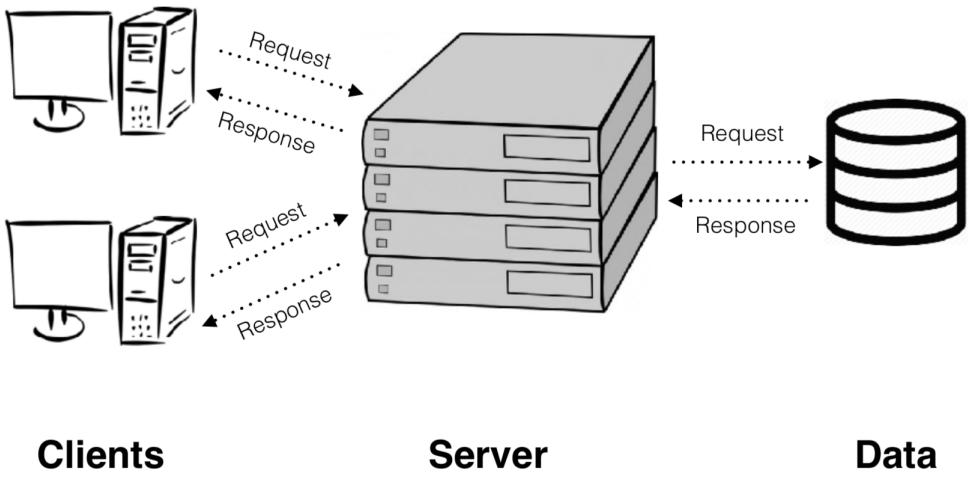


Figure 5: Server-Client Communication

11 Sockets

11.1 Pipe

1-way communication

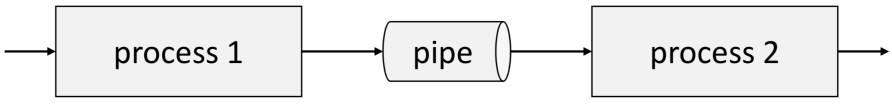


Figure 6: 1 Way Communication

2-way communication

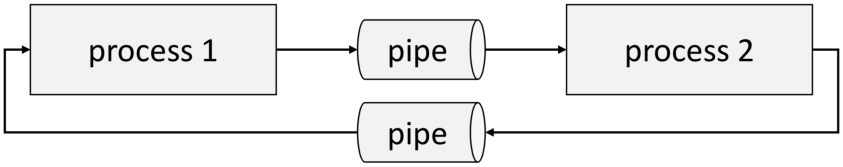


Figure 7: 2 Way Communication

11.2 Sockets

Sockets are one way to implement 2-way channels. The TCP protocol connects sockets on different machines across the internet.

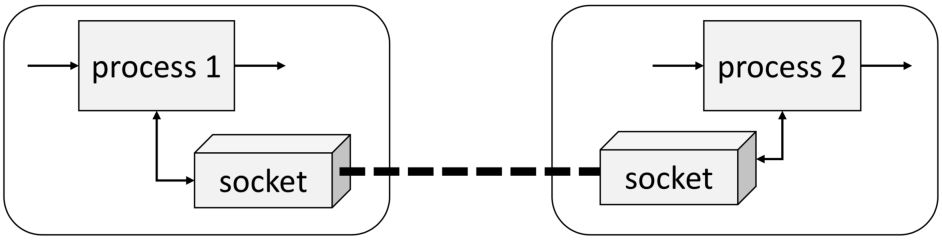


Figure 8: TCP socket

socket states:

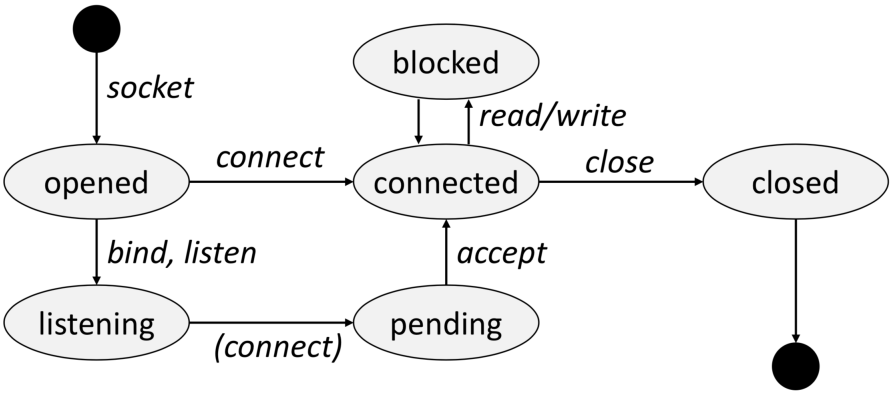


Figure 9: Socket States

File descriptor: A file descriptor (fd), like a FILE*, is a way of referring to a file that the operating system has opened

Library

- **int socket(int domain, int type, int protocol)** (*sys/socket.h*) creates a new socket. While *domain=AF_INET* refers to the IPv4 and *type=SOCK_STREAM* means using the TCP/IP protocol. The return value is a file descriptor unless fail (*ret=-1*).
- **int bind(int fd, struct sockaddr *addr, socklen_t len)** (*sys/socket.h*) binds the socket to the specific port, while the *fd* refers to the socket and *addr* represents the port.
- **int listen(int fd, int backlog)** (*sys/socket.h*) listens the socket to wait the new request.

- **int accept(int fd, struct sockaddr *addr, socklen_t *len)** (*sys/socket.h*) will accept request from the socket, the return value is a new fd for the connection unless fail (*ret=-1*).
- **int close(int fd)** (*unistd.h*) close the file descriptor.
- **int poll(struct pollfd[] fds, nfds_t n, int timeout)** (*poll.h*) poll the file descriptors.

12 Transactions

A transaction is an action or sequence of actions that needs to be performed either fully or not at all. Many database servers offer ACID transactions:

- A is for Atomicity.
- C is for Consistency.
- I is for Isolation.
- D is for Durability.