COMP20200 Unix Programming

Lecture 17

Ravi Reddy Manumachu (ravi.manumachu@ucd.ie)

School of Computer Science, University College Dublin, Ireland

05/04/2023

Lecture overview

- Review re-entrancy, async-signal-safety, and thread-safety.
- Develop multithreaded TCP socket server using pthreads.
- Compiling and running the client and server applications.

Reentrant Functions: Local Variables

- A function is said to be **reentrant**:
  - If it can be interrupted in its execution,
  - And then safely called again in a different context,
  - Before the previous invocations complete execution.
- For example, a function that is interrupted in a main program (context 1) and invoked in a signal handler (context 2).
- Main program and signal handler provide two independent (and not simultaneous) contexts of execution.
- A function that uses only local variables is guaranteed to be reentrant.

Reentrant Functions: Global Variables

- A function may be nonreentrant if it updates global data structures.
- For example: *malloc()* maintains a linked list of freed memory blocks available for reallocation from the heap.
- If a call to *malloc()* in the main program is interrupted by a signal handler that also calls malloc(), then this linked list can be corrupted.
- Therefore, the malloc() family of functions are *nonreentrant*.

Reentrant Functions: Static Variables

- Library functions that return information using statically allocated memory are *nonreentrant*.
- Static data structures are those declared with *static* keyword globally or locally inside a function.
- Examples: crypt(), getpwnam(), gethostbyname(), and getservbyname().

async-signal-safe functions

- An async-signal-safe function is one that the implementation guarantees to be safe when called from a signal handler.
- List of async-signal-safe functions.

```
shell> man signal-safety
SIGNAL-SAFETY(7)           Linux Programmer's Manual          SIGNAL-SAFETY(7)

NAME
       signal-safety - async-signal-safe functions
...
       Function                 Notes
       abort(3)                 Added in POSIX.1-2003
       ...
       fork(2)                  See notes below
       ...
       read(2)
       ...
       waitpid(2)
       ...
       write(2)
```

Reentrant Functions and Thread Safety

- A function is said to be **thread-safe** if it can safely be invoked by multiple threads at the same time.
- A reentrant function can achieve thread safety without the use of mutexes.
- It can do this by avoiding the use of global and static variables.
- For example, a function *f()* creates a local buffer and passes it to *g()*.
- *g()* manipulates the contents of the buffer and returns it to *f()*.
- *g()* is reentrant and thread-safe.

Reentrant Functions

- For many functions that are nonreentrant, a reentrant function may be available with a suffix _r.
- Look at a man page of a function for this information.

Is our SIGCHLD signal handler reentrant and thread safe?

```c
static void
reapChild(int sig)
{
  int savedErrno = errno;
  while (waitpid(-1, NULL,
      WNOHANG) > 0)
    continue;
  errno = savedErrno;
}
```

- It is not thread-safe since it updates a global data structure (*errno*) without taking locks.
- However, signal handlers are not meant by to be called by multiple threads of execution.

TCP Socket Server using Pthreads

Multithreaded servers

We look at two variations.

- Server using detached threads to handle client requests simultaneously.
- Server using joinable threads to handle predefined number of clients simultaneously.

Multithreaded servers

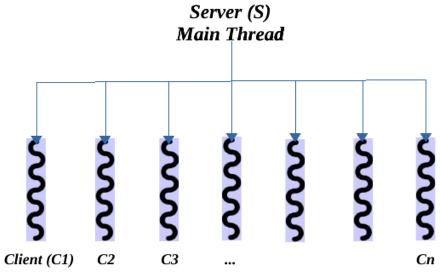

Figure: Server using detached threads.

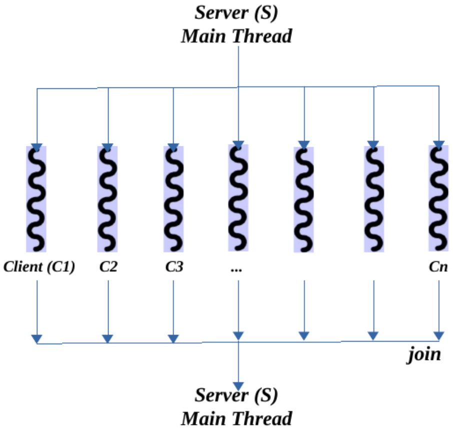

Figure: Server using joinable threads.

Multithreaded server

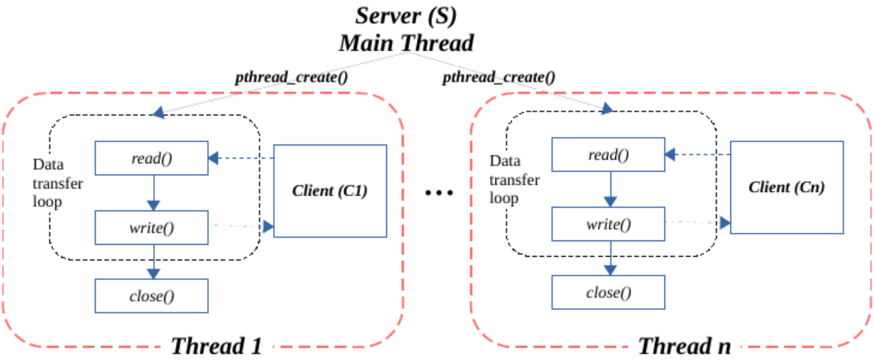

Figure: Multithreaded server servicing clients using threads.

- The *for loop* in the server will involve creation of threads to handle clients concurrently.

Server code: Using Pthreads

- SIGCHLD signal handler setup not required.
- In the server loop, the server's main thread creates a thread using *pthread_create* to handle a client request.
- **Revisit lecture 11 on threads.**

Server code: Using detached threads

```
pthread_attr_t tattr;
pthread_attr_init(&tattr);
pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
for (;;) /* Main server for loop */
{
    ...
    pthread_t t;
    pthread_create(&t, &tattr, handleRequest, &cfd);
}
pthread_attr_destroy(&tattr);
```

- The main thread creates a detached thread to handle a client request using *pthread_create()* call.

Server code: Using detached threads

```
pthread_attr_t tattr;
pthread_attr_init(&tattr);
pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
for (;;) /* Main server for loop */
{
    ...
    pthread_t t;
    pthread_create(&t, &tattr, handleRequest, &cfd);
}
pthread_attr_destroy(&tattr);
```

- To create a detached thread, the detached thread attribute must be passed during the thread creation.
- This attribute is created using PTHREAD_CREATE_DETACHED flag.

Server code: Using detached threads

```
pthread_attr_t tattr;
pthread_attr_init(&tattr);
pthread_attr_setdetachstate(&tattr,PTHREAD_CREATE_DETACHED);
for (;;) /* Main server for loop */
{
    ...
    pthread_t t;
    pthread_create(&t, &tattr, handleRequest, &cfd);
}
pthread_attr_destroy(&tattr);
```

- The attribute is destroyed using *pthread_attr_destroy()* call.

Server code: Thread creation

```
for (;;)
{
  int cfd = accept(lfd, NULL, NULL);
  if (cfd == -1)
    continue; /* Exit or Continue */
  int* arg = (int*)malloc(sizeof(int));
  *arg = cfd;
  pthread_t t;
  pthread_create(&t, &tattr, handleRequest, arg);
}
```

- Thread is created using *pthread_create()* call.
- The input arguments are:
  - Thread attribute (*tattr*).
  - The thread function (*handleRequest*).
  - The input data to the function (*cfd*).

Server code: Thread creation

```
for (;;)
{
  int cfd = accept(lfd, NULL, NULL);
  if (cfd == -1)
    continue;
  int* arg = (int*)malloc(sizeof(int));
  *arg = cfd;
  pthread_t t;
  pthread_create(&t, &tattr, handleRequest, arg);
}
```

- The output argument is the thread ID (t).
- The connection fd (*cfd*) is passed as input to the thread function.

Server code: handleRequest() function

```
void*
handleRequest(void* input)
{
  int cfd = *(int*)input;
  /* A read followed by a write */
  free(input);
  pthread_exit(NULL);
}
```

- In the *handleRequest()* thread function, the thread obtains connection fd (*cfd*) that is passed as input.
- The read and write logic is the same as the iterative case.
- *pthread_exit()* call terminates the thread.

Multithreaded Server Using Joinable Pthreads

Multithreaded Server Using Joinable Pthreads

```c
1 pthread_t tid[NUMCLIENTS];
2 for (;;)
3 {
4     if (nClients == NUMCLIENTS) {
5        printf("Serviced %d clients. Exiting.\n", NUMCLIENTS);
6        break;
7     }
8     pthread_create(&tid[nClients], NULL, handleRequest, arg);
9     nClients++;
10 }
```

- The server deals with a fixed number of clients, *NUMCLIENTS*.
- In Line 1, the main thread creates an array of thread IDs having size equal to the number of clients.

Multithreaded Server Using Joinable Pthreads

```
1 pthread_t tid[NUMCLIENTS];
2 for (;;)
3 {
4     if (nClients == NUMCLIENTS) {
5         printf("Serviced %d clients. Exiting.\n", NUMCLIENTS);
6         break;
7     }
8     pthread_create(&tid[nClients], NULL, handleRequest, arg);
9     nClients++;
10 }
```

- In Line 8, the main thread creates a thread (*tid[t]*) to deal with the client, *nClients*.

Multithreaded Server Using Joinable Pthreads

```
for (;;)
{
    int cfd = accept(lfd, NULL, NULL);
    int* argtot = (int*)malloc(sizeof(int));
    *argtot = cfd;
    pthread_create(&tid[nClients], NULL, handleRequest, argtot);
}
```

- The server's main thread accepts a client connection.
- It passes the connection fd (*cfd*) to the newly created thread.

Server code: handleRequest() function

```
void*
handleRequest(void* input)
{
  int cfd = *(int*)input;
  /* A read followed by a write */
  free(input);
  pthread_exit(NULL);
}
```

- The thread function is the same as before.
- It obtains the connection fd (*cfd*) that is passed as input to the thread function.
- The read and write codes are the same as the iterative case.
- *pthread_exit()* call terminates the thread.
- The argument passed to *pthread_exit()* call is available in the main thread in the *pthread_join()* call.

Multithreaded Server Using Joinable Pthreads

```
1 pthread_t tid[NUMCLIENTS];
2 for (;;)
3 {
4 ...
5     pthread_create(&tid[nClients], NULL, handleRequest, arg);
6     nClients++;
7 }
8 for (t = 0; t < NUMCLIENTS; t++)
9     pthread_join(tid[t], NULL);
```

- Once the server has serviced all the clients, it can now join with all the threads.
- The main thread issues *pthread_join()* to join with thread, *tid[t]* (Line 9).

Multithreaded Server Using Joinable Pthreads

```
1 pthread_t tid[NUMCLIENTS];
2 for (;;)
3 {
4 ...
5     pthread_create(&tid[nClients], NULL, handleRequest, arg);
6     nClients++;
7 }
8 for (t = 0; t < NUMCLIENTS; t++)
9     pthread_join(tid[t], NULL);
```

- The main thread can get the status of the thread (*tid[t]*) in the second argument to *pthread_join()*.
- The loop (Lines 8-9) becomes a synchronization point since the main thread blocks until it has joined with all the threads.

Multithreaded Server Using Threads: Summary

- The key differences from the server using parallel processes are
  - No SIGCHLD signal handler.
  - Using threads instead of processes to service clients.
  - The data transfer loop to service a client takes place in the thread's *handleRequest()* function.
- Rest of the code is completely reusable.

Client-server application: Compilation

- mtserver1.c - Contains the server code using parallel processes.
- mtserver2.c - Contains the multithreaded server code using detached pthreads.
- mtserver_join.c - Contains the multithreaded server code using joinable threads.
- iclient.c - Contains the client code.

```
$ gcc -o iclient iclient.c
$ gcc -o mtserver1 mtserver1.c
$ gcc -o mtserver2 mtserver2.c -lpthread
$ gcc -o mtserver_join mtserver_join.c -lpthread
```

Client-server application: Execution

In one terminal,

```
$ ./mtserver2 12222
Listening on (127.0.0.1, 12222)
Connection from (localhost, 43582)
Received BBRMZQBEYDHTKMHQROGZCHBVRXWBBKK
Connection from (localhost, 43584)
Received BBRMZQBEYDHTKMHQROGZCHBVRXWBBKK
...
```

In a different terminal,

```
$ for i in {1..25}; do (./iclient 127.0.0.1 12222 &); done
Sending BBRMZQBEYDHTKMHQROGZCHBVRXWBBKK to 127.0.0.1:12222
Received BBRMZQBEYDHTKMHQROGZCHBVRXWBBKK
...
```

Executing clients simultaneously

```
$ for i in {1..25}; do (./iclient 127.0.0.1 12222 &); done
```

- This shell command invocation executes 25 clients simultaneously.
- The execution (./iclient 127.0.0.1 &) means that the client is executed as a background process
- More on this in lectures on bash scripting language.

Lookahead: Lecture 18

Sockets is an IPC tool. We overview IPC in the next lecture.

Q & A