SPECIAL TOPICS

CS2141 - Software Development using C/C++

Topics

- Checking Files Exist
- Serial Communications
- Multithreading

Checking File Existence

- Could just open file, check return code, and close
- Leads to false negatives if there are permissions errors
- `stat()` is part of the C standard libraries
- Attempts to get file attributes, not open the file
- Will return 0 if successful, something else otherwise
- include `sys/stat.h` to use

stat() Example

```
#include <sys/stat.h>

bool FileExists(string strFilename) {
  struct stat stFileInfo;
  bool blnReturn;
  int intStat;

  intStat = stat(strFilename.c_str(),&stFileInfo);
  if(intStat == 0) {
    blnReturn = true;
  }
  else {
    blnReturn = false;
  }

  return(blnReturn);
}
```

Other stat() Goodies

- stat() provides a bunch of other information:
  - File owner/group
  - File permissions
  - Access / creation times
  - File system details

Serial Communications

- Each port gets a “file” in the file system
- Linux provides `/dev/ttyS0`, `/dev/ttyS1`...
- Files may have restricted permissions, check if you need them
- Can read/write serial port as though it were a file
- Should use C functions (`open`/`read`/`write`)
- Use `termios.h` to configure the port as needed

Opening a Serial Port

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <termios.h>

int open_port(void) {
    int fd; /* File descriptor for the port */

    fd = open("/dev/ttyS0", O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd == -1) {
        perror("open_port: Unable to open /dev/ttyS0 - ");
    }
    else
        fcntl(fd, F_SETFL, 0);

    return (fd);
}
```

Reading & Writing

- Writing Data:
  ```
  n = write(fd, "HELLO", 5);
  if (n < 0)
      fputs("write() of 5 bytes failed!\n", stderr);
  ```
- By default, read will block until data is ready
- Use `fcntl(fd, F_SETFL, FNDELAY);` to return immediately
- See link on website for more details

Multithreading

- Techniques for allowing a program to do two things at once
- Can be used to separate long-running tasks from main program
- Very common in GUI design
    - Interface in one thread, logic in another
- Threads share access to resources (global vars, open files, etc)
- Introduces many unusual and hard-to-find bugs

pthreads

- POSIX threading library for C, lives in `pthread.h`
- Works by starting a new thread with a function call
- Provides reasonable support for thread synchronization
- Use `-lpthread` flag at compilation

Managing Threads

- Creating a thread:

```
int pthread_create(pthread_t * thread,
        const pthread_attr_t * attr,
        void * (*start_routine)(void *),
        void *arg);
```

- Joining a thread (waiting for it to return):

```
int pthread_join(pthread_t th, void **thread_return);
```

- Exiting a thread:

```
void pthread_exit(void *retval);
```

pthreads Example

```
#include <stdio.h> #include <stdlib.h> #include <pthread.h>

void *print_message_function( void *ptr ){
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}

int main(){
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;
     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*)
message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*)
message2);
     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}
```

Synchronizing Threads

- Threads run independently of each other
- Critical sections of code must be run by no more than one thread at a time
- Use mutexes to control access to critical sections

Mutexes

- Define a mutex somewhere all threads can see it:
  `pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;`
- Lock the mutex before critical section:
  `pthread_mutex_lock( &mutex1 );`
- Unlock mutex after the critical section:
  `pthread_mutex_unlock( &mutex1 );`
- First thread gets mutex lock. All others block until lock frees

Thread Pitfalls

- Race Conditions
  - Threads may not run all at once or in the order created
- Thread safety
  - Avoid static or global variables that may be clobbered
- Mutex Deadlock
  - Always be sure to unlock mutexes when done with them