CS 33: Introduction to Computer Organization

Week 8: Part 2

Command Line Args

- One question on the homework asks you to write program that takes in command line arguments.
- Command line arguments are arguments that you specify when executing the program:
  - “./program arg1 arg2 arg3”

Command Line Args

- Declare the main function as follows:
  - int main(int argc, char * argv[]) or int main(argc, char ** argv)
- int argc corresponds to how many arguments are included.
- char * argv[] is an array of pointers to C-strings. These correspond to the arguments.

Command Line Args

- Consider:
  - ./program arg1 arg2 arg3
  - argc is 4
  - argv[0] is “./program\0”
  - argv[1] is “arg1”
  - argv[2] is “arg2”
  - Hey kids, can you help Slylock Fox figure out what argv[3] corresponds to?

Command Line Args

- Convert a string to an integer:
  - char * blah = “1234”;
  - atoi(blah) is an int with value 1234

Intro to Deadlocks

- The major theme of the second half of Chapter 12 is synchronization.
- That is, in order to coordinate multiple threads/processes, there are times in which we must wait in order to ensure a particular ordering.
- This is accomplished via P(&s) (sem_wait), V(&s) (sem_post), pthread_join, wait_pid, etc.
- We are implementing behavior where the execution of a program is unconditionally halted.

Intro to Deadlocks

- Consider the following code:
-
```
void* run_wait(void * arg)
{
  printf("PEER THREAD RUNNING\n");
}

int main()
{
  pthread_t tid;
  pthread_create(&tid, 0, run_wait, NULL);
  printf("MAIN THREAD RUNNING\n");
}
```

- When I run this, “PEER THREAD RUNNING” is never printed. What's going on?

Intro to Deadlocks

- It seems that given that the main function exits so quickly, the main thread exits before the helper thread can do any of it's work.
- Upon completion it looks like the main thread is killing the helper thread?
- Is this a job for pthread_detach?

Intro to Deadlocks

- Consider the following code:

-
```
void* run_wait(void * arg)
{
  pthread_detach(pthread_self());
  printf("PEER THREAD RUNNING\n");
}

int main()
{
  pthread_t tid;
  pthread_create(&tid, 0, run_wait, NULL);
  printf("MAIN THREAD RUNNING\n");
}
```

Intro to Deadlocks

- The same race condition exists.
- If the peer thread never got around to printing, it may not get around to detaching.
- Let's force the thread to be detached.

Intro to Deadlocks

- Consider the following code:

-
```
void* run_wait(void * arg)
{
  printf("PEER THREAD RUNNING\n");
}

int main()
{
  pthread_t tid;
  pthread_create(&tid, 0, run_wait, NULL);
  prthread_detach(tid);
  printf("MAIN THREAD RUNNING\n");
}
```

Intro to Deadlocks

- Still... this only prints out “MAIN THREAD RUNNING”.
- This is beginning to seem like an exercise in futility.
- The fact is, despite the notion that threads are peers and equals, the main thread is just a little more equal than the rest.
- This is because when the main thread completes, the main process completes.

Intro to Deadlocks

- Since threads are shared among a single processes, ending the process that threads are running on ends all threads.
- If you want to guarantee that all launched threads are completed before executing, you must have a pthread_join.
- This can also be resolved by having the main thread calls pthread_exit(0), which will wait for all peer threads to exit before continuing.

Deadlocks

- Consider the case where we have two shared variables and if we want to access them, we must first sem_wait for a semaphore.
- We have a thread function that reads from the shared variables and prints them out.
- In the mean time, some other thread is responsible for writing to the variables.
- Only one thread should be reading or writing to the variables.

Deadlocks

```
int main()
{
  sem_init(&t, 0, 1);
  sem_init(&u, 0, 1);
  pthread_t tid;
  pthread_create(&tid, 0, run_wait,
NULL);
  printf("MAIN THREAD
RUNNING\n");

  sem_wait(&u);
  sem_wait(&t);
  shared_t = 0;
  shared_u = 1;
  sem_post(&t);
  sem_post(&u);

  void * ret;
  pthread_join(tid, &ret);
}
```

```
sem_t t;
sem_t u;

char shared_t = 0x74;
char shared_u = 0x75;

void* run_wait(void * arg)
{
  printf("PEER THREAD
RUNNING\n");
  sem_wait(&t);
  sem_wait(&u);
  printf("%d\n", shared_t);
  printf("%d\n", shared_u);
  sem_post(&u);
  sem_post(&t);
}
```

Deadlocks

- This certainly seems to accomplish the goal.
- If you run it, it usually works.
- ...except that it might not.
- Remember that even though there is a certain behavior that we expect when threads are running, we can make no assumptions about it when considering correctness.

Deadlocks

- The main thread:
  - sem_wait(&u) then sem_wait(&t)
- The helper thread:
  - sem_wait(&t) then sem_wait(&u)
- What if the order of instruction executions is as follows:
  - 1. main: sem_wait(&u)
  - 2. helper: sem_wait(&t)
  - 3. Anything else.

Deadlocks

- The main thread will attempt to get a hold of t which the helper currently has. Meanwhile, the helper will try to get a hold of u, which the main thread has.
- Both are blocked indefinitely and neither can continue.
- This is the quintessential deadlock case.

Deadlocks

- The case in which deadlock can occur:
  - There is a cyclic dependency of locks where one thread holds lock A and waits for lock B while another thread holds lock B and waits for lock A.
  - T1 holds L1 and waits for L2, T2 holds L2 and waits for L3, T3 …, TN holds LN and waits for L1.

Deadlocks

- According to the book 2nd ed. (pg. 987), a way to guarantee a deadlock free program is to follow the simple rule where for any pair of locks, if there are any threads that acquire both, then they must acquire them in the same order.

Deadlocks

- According to the book 2nd ed. (pg. 987), a way to guarantee a deadlock free program is to follow the simple rule where for any pair of locks, if there are any threads that acquire both, then they must acquire them in the same order.
- ...and I even believed that for a little while.

Deadlocks

- However, in my previous statement:
  - T1 holds L1 and waits for L2, T2 holds L2 and waits for L3, T3 …, TN holds LN and waits for L1.
- ...that's a deadlock case in which no pair of deadlocks are held by multiple threads.
- The revised rule (on the CSAPP 2nd Edition Errata listing): define a total ordering for the locks, make sure that when acquiring, the total ordering is followed.
- Thus, in this case, if TN is forced to acquire L1 before LN, deadlock won't occur

Thread Safety

- The book defines thread safety quite thoroughly (if a little confusingly).
- A function is “thread-safe” if it will be correct even if called repeatedly by multiple threads.
- Functions that are thread-unsafe fall in one of four categories:
- Class 1: Functions that don't protect shared variables.

Thread Safety

- Class 2: Functions that keep state across multiple invocations (functions whose current result depends on previous invocations)
- Class 3: Functions that return pointers to static variables.
- Class 4: Functions that call class 2 thread unsafe functions and functions that call class 1 and 3 thread unsafe functions and don't protect the function calls (with synchronization).

Thread Safety

- Aside: What does the static keyword mean?
- If applied to a global variable this means that this variable is visible only to the module in which it's located.
- Ex. if I have a project that includes main.c and blah.c and blah.c defines a static global variable called “foo”, that instance of foo is not visible to main.c

Thread Safety

- If applied to a local variable, static means across all threads and invocations of that function, there is only one instance of that variable.
- Consider:

```
int foo()
{
  static int i = 0;
  i += 1;
}
```

Thread Safety

```
int foo()
{
  static int i = 0;
  i += 1;
}
```

- The first thread to call this will initialize int i. From then on, all calls to foo will refer to this singular int.
- If thread 1 calls foo, i will be incremented. If thread 2 calls foo, it will find that i = 1 and it will increment it.

Thread Safety

- Which one of these “static” variables is the one that is referred to by this “thread unsafe” case 3?

Thread Safety

- Which one of these “static” variables is the one that is referred to by this “thread unsafe” case 3?
- Probably both right? Static globals will be shared among threads in a module (this has nothing to do with the static keyword) and static locals will be shared among threads calling the function.

Thread Safety

- Consider the ctime function which converts a time to a string:
- char * ctime(const time_t * timer)
- time_t is a data type that is essentially (but not quite) a number that corresponds to the time since the epoch.
- The return value is a C-string that is in a readable format.

Thread Safety

- The problem is that the pointer that ctime returns is a static pointer to a special location.
- Therefore, this is class 3 thread-unsafe.
- Ex. if two threads call ctime in quick succession, both will modify the data that the pointer points to. If the second call to ctime modifies the pointer before the first call can read from it, then the first call to ctime will return the same string as the second call.

Reentrancy

- Reentrancy is defined as a function that doesn't use any shared variable at all.
- This is an incredibly simple definition that tells us exactly nothing about what “reentrancy” really means.
- However, this is probably the definition you can fall back on when doing the homework.

Reentrancy

- Reentrancy is a concept that predates multi-threading.
- Essentially, a re-entrant function is one that can be interrupted by a signal and then re-entered safely... *all from within the same thread*.
- By safely, we mean that the result will be correct in terms of value and in terms of execution behavior.
- How can this “re-entering” behavior happen?

Interrupts

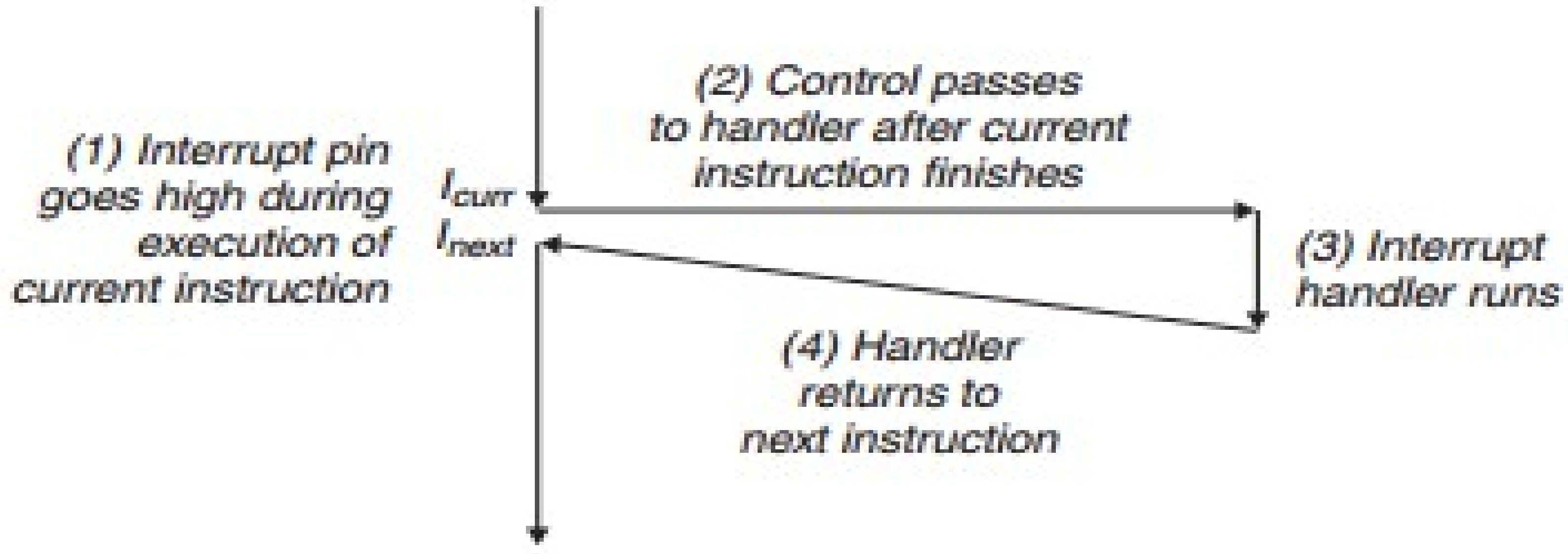
- Sometimes when your program is running, it has to be able to respond to outside stimuli.
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
- These signals will be sent to running programs.
- Asynchronous
  - Occurs independently of currently executing program

Interrupt Handling

- I/O device triggers the “interrupt pin”
- After current instruction, stop executing current thread of instructions and control switches to interrupt handler.

Interrupt Handling

- Interrupt handler handles interrupt.
- Control is given back to previously executing thread of instructions.
- Previous program executes the next instruction.


(1) Interrupt pin goes high during execution of current instruction
I_{curr}
I_{next}
(2) Control passes to handler after current instruction finishes
(3) Interrupt handler runs
(4) Handler returns to next instruction

CLARIFICATION

- The following example does not make a clear distinction between “interrupt” and “signal” but in reality, there is an important difference.
- To clarify, interrupts and exception handlers are based on the actions and events of the hardware. When an exception occurs (ex. processor tries to divide by zero), control switches to kernel mode and the kernel executes the exception handler. If the kernel handles the exception, processes may not automatically be aware of the exception.
- Signals are the high level mechanism of making these low level hardware exceptions known to software. If an interrupt were to occur, the kernel would execute the appropriate exception handler. Additionally however, the kernel sends a “signal” to the relevant processes.
- When the processes receive the signal, they respond by executing the signal handler, in *user mode*, but with the same thread that was paused.
- As a result, the signal handler is not the same as the exception handler, but an exception may cause the signal handler of a process to be run.

Reentrancy

- Thus, if a program is running a particular thread and that thread is in a function, it is possible that a signal causes the thread's execution to be switched to different code to handle the signal.
- If the signal handler calls the same function that you were in when you were interrupted (by the signal), you better hope that function was reentrant.
- Consider ctime again.

Reentrancy

- Pretend that ctime has some code that looks like this (char* ptr is shared):

```
…
  strcpy(ptr, local_ptr); // Copies the string from
  local_val = 10;         //  local_ptr to ptr
  return ptr;
}
```

Reentrancy

- Say in our thread, we have just finished executing the strcpy and local_ptr = “current”

```
…
  strcpy(ptr, local_ptr);
  local_var = 10;        ← Current, ptr = “current”
  return ptr;
}
```

- Now, a signal is received which causes the thread to run the signal handler (with the intent of returning to the “Current” line once the signal is handled)

Reentrancy

- This thread should be returning ptr which contains the string we just copied “current”.
- What if the signal handler calls ctime?

Reentrancy

- …
  ```
    strcpy(ptr, local_ptr); ← Signal Handler
    local_var = 10;         ← Current, ptr = “current”
    return ptr;
  }
  ```

- Say the signal handler calls ctime again except now local_ptr = “signal”.
- The signal handler copies local_ptr to ptr, returns and completes.

Reentrancy

- …
  ```
    strcpy(ptr, local_ptr);
    local_var = 10;       ← Current, ptr = “signal”
    return ptr; ← Signal Handler
  }
  ```
- When the signal handler completes, the thread goes back to the instruction that it would have executed before the signal.

Reentrancy

- …
  ```
    strcpy(ptr, local_ptr);
    local_var = 10;        ← Current, ptr = “signal”
    return ptr;
  }
  ```

- Now the result is wrong.

Reentrancy

- The solution for class 3 thread-unsafe functions is as follows:

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    sem_wait(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp); /* Copy string from shared to private */
    sem_post(&mutex);
    return privatep;
}
```

Reentrancy

- This wraps the call to ctime in a lock that means only one thread can access a call to ctime.
- The locked critical section will copy the string pointed to by the static pointer of ctime and copy it into a local non-shared pointer.
- Thus, one thread's call to ctime_ts cannot be affected by another thread's call to ctime_ts.

Reentrancy

- However, as Practice Problem 12.12 suggests, this is non-reentrant. Why?
- The book's answer:
  - “The ctime_ts function is not reentrant because each invocation shares the same static variable returned by the ctime function. “
  - ...and the award for least helpful answer goes to...

Reentrancy

- Exactly why is it such a problem if a single thread is interrupted and ctime_ts is called again?
- Consider the following flow of execution:

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);  ← Current
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);
    sharedp = ctime(timep); ← Current, mutex = 0
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

- SIGNAL!
- Signal handler calls ctime_ts.

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp; ← Signal

    P(&mutex);
    sharedp = ctime(timep); ← Current, mutex = 0
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

Reentrancy

```
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;

    P(&mutex);  ← Signal
    sharedp = ctime(timep); ← Current, mutex = 0
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```

- When the signal handler reaches P, it must wait until the mutex is released.

Reentrancy

- But it will never be released. This isn't a multithreaded context in which context can switch to the original execution.
- This thread IS the original thread that acquired the lock.
- This is a case of deadlock caused by one thread waiting on itself.

End of

The Eighth Week

-Two Weeks Remain-