CS 33 Discussion 8

May 25, 2017

Agenda

Midterm Discussion

Synchronization

Midterm Statistics

Mean: 56

Median: 58

Question	TA
1a,b,c	Anurag Pande
1d,e	Jin Wang
2a	Aanchal Dalmia
2b	Mingda Li
2c	Nazanin Farahpour
3	Nikita Tulpule
4	Shikhar Malhotra
5a,b,c	Harshada Wadekar
5d	Swathi Patnaikuni

Q1

1a (6 minutes). Write two C functions with the following APIs:

```
unsigned f2u (float x);
float u2f (unsigned y);
```

f2u(X) should yield an unsigned integer Y that has the same 32-bit representation as X. For example, f2u(-0.1f) should yield 0xbdcccccd, because -0.1f has a sign bit 1, an exponent field 0x7b, and a fraction field 0x4ccccd, and ((1u<<31) | (0x7b<<23) | 0x4ccccd) == 0xbdcccccd. The u2f function should be the reverse operation, i.e., f2u(u2f(Y)) == Y should be true for all unsigned values Y.

1b (3 minutes). Does the C expression u2f(f2u(X)) == X yield 1 for all float values X? If so, briefly justify why; if not, give a counterexample.

1c. Give two unsigned values Y1 and Y2 such that the C expression (Y1!=Y2 && u2f(Y1) == u2f(Y2)) yields 1.

Answer 1a

```
unsigned f2u(float x){
        union u{
                float f;
                unsigned u;
        };
        union u u1;
        u1.f = x;
        return u1.u;
}
```

```
unsigned u2f(unsigned y){
        union u{
                float f;
                unsigned u;
        };
        union u u1;
        u1.u = y;
        return u1.f;
}
```

Answer 1b

If X is a NaN, then the expression will evaluate to 0.

Hence, No.

Answer 1c

Y1 = +0

Y2 = -0 (or 0x80000000 or 2^31 or 1<<31)

Q1d,e

1d (4 minutes). Which of the following machine-language functions, if any, are plausible x86-64 implementations of f2u and of u2f, respectively?

```
A: movl  %edi, -4(%rsp)
   movss -4(%rsp), %xmm0
   ret

B: movl  %edi, 4(%rsp)
   movss 4(%rsp), %xmm0
   ret

C: movss %xmm0, -4(%rsp)
   movl  -4(%rsp), %eax
   ret

D: pxor      %xmm0, %xmm0
   movl      %edi, %edi
   cvtsi2ssq %rdi, %xmm0
   ret

E: cvttss2siq %xmm0, %rax
   ret

F: movd %xmm0, %eax
   ret
```

1e (9 minutes). For each machine-language function (A)-(F) that is not a valid implementation of either u2f or f2u, explain why not. If there is some other C-language function that the machine-language function is a valid implementation of, give such a function; if not, explain why not.

Answer 1d

u2f: **A** (2 points); other answer (0 point)
f2u: **C,F** (2 points); C (1 point); F (1 point); other answer (0 point)

Answer 1e

B: invalid, you're trashing your return address by moving stuff there

D: float D(unsigned x){return x;}

E: unsigned E(float t) {return t;}

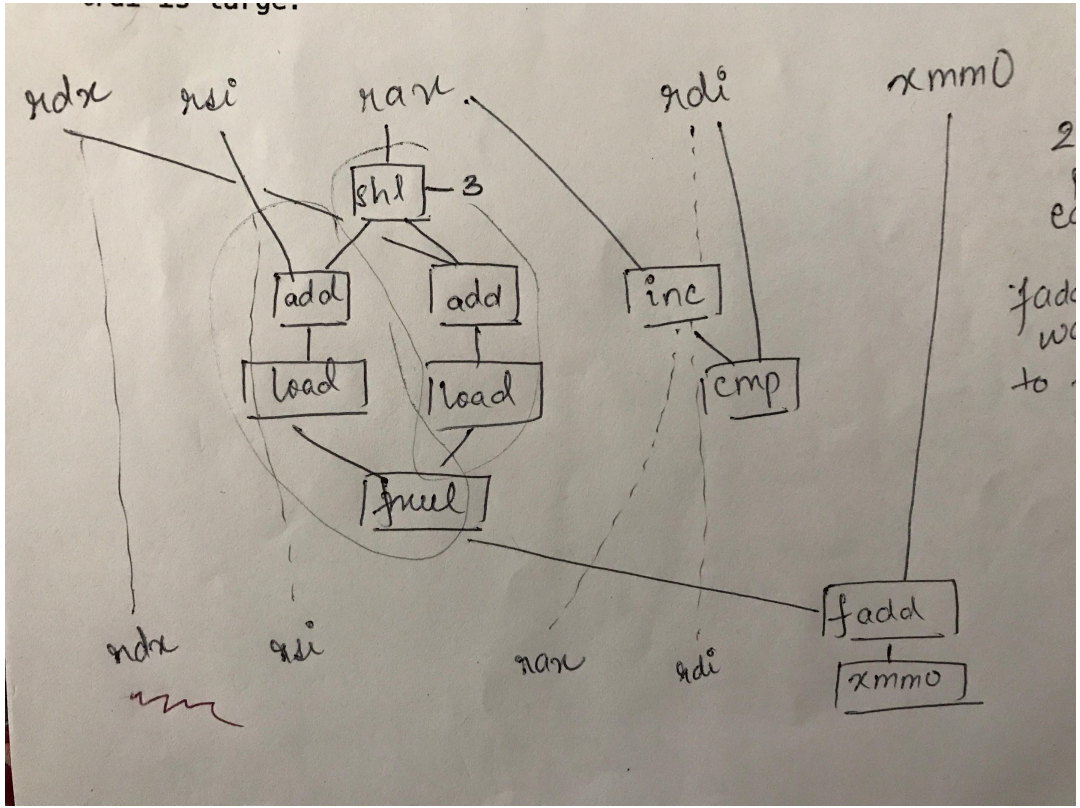
Q2a

Consider the following x86-64 function foo:

```
foo:
        testq %rdi, %rdi
        je    .L4
        xorpd %xmm0, %xmm0
        xorl  %eax, %eax
.L3:
        movsd (%rsi,%rax,8), %xmm1
        mulsd (%rdx,%rax,8), %xmm1
        incq  %rax
        cmpq  %rax, %rdi
        addsd %xmm1, %xmm0
        jne   .L3
        ret
.L4:
        xorpd %xmm0, %xmm0
        ret
```

Construct a data-flow representation of the micro-operations for the inner loop of the function foo. Identify any critical paths that are likely to be a performance bottleneck when %rdi is large.

Answer 2a


rdx
rsi
rax
rdi
xmm0
shl
3
add
add
inc
load
load
cmp
fmul
fadd
xmm0
rdx
rsi
rax
rdi

Q2b

2b (11 minutes). Suppose the function foo is executing on a Kaby Lake processor. Kaby Lake processors have the same set of functional units as the Haswell. How well will the inner loop be parallelized on this processor, using instruction-level paralellism? Briefly justify your answer by appealing to your answer to the previous subquestion.

Answer 2b

Kaby Lake has 2 FPUs and that is the bottleneck (see dataflow diagram).
Do fmul and fadd of two consecutive iterations in parallel.

Q2c

2c (11 minutes). Suppose you have several chips that implement the same x86-64 instruction set. Each chip has just one cache for RAM. The cheapest chip uses a direct-mapped cache; the next-cheapest one uses a 2-way set-associative cache; the next-cheapest one a 4-way set-associative cache, and so on for 8-way and 16-way. All the caches use a 64-byte cache line and they all contain 1 MiB of data. If you are interested in executing the function foo on many small arrays (with typical size 256 bytes), which of these chips will you be most interested in buying, if you want to maximize the bang for your buck? Briefly justify your answer.

Answer 2c

In direct-mapped, collision = crash
2-way : if both arrays map to the same location, the 2 load boxes (in the dataflow diagram) will not collide.

Q3

3 (11 minutes). For each of the following forms of machine parallelism, give an example of an application that will likely work better with this form of parallelism than with any of the other forms listed. Briefly justify your answers.

Instruction level parallelism
Multiplexing
Process parallelism
SIMD
Thread parallelism

Answer 3

Multiplexing: Good for activities with delays. For example, applications which need to wait for user inputs.

Process Parallelism: Good for loosely coupled systems. Different parts run independently.

Thread Parallelism: Good for systems where different parts share information like in web servers. (For process and thread you were expected to give any logical example with explanation of why process or thread suits it better)

SIMD: Single instruction multiple data. Works best for operations on arrays, matrices where same instruction has to be performed on all data. ML is an application.

Instruction Level Parallelism: Can be used in most cases for optimisation where there are less data dependencies and not too many branches. (This is useful whenever the other four are not.)

Q4

4 (10 minutes). Suppose you are designing a computer with two caches: a smaller L1 cache (one per core), and a larger L2 cache (shared among all the cores). You are trying to decide whether the caches should be *exclusive*, i.e., a data word is never in both L1 and L2, or *strictly inclusive*, i.e., every data word cached in L1 is also cached in L2. Give pros and cons of both approaches.

Answer 4

Pros for exclusive: More capacity, hence more data can be cached; Less redundancy of data; Holds one copy for each data value

Cons for exclusive: More management to keep track of uniqueness; Different cache lines are difficult to support; Have to search L1 and L2 to get hold of specific data

Pros for inclusive: Simpler in design; Different cache line sizes are supported (like 16 for L1 and 64 for L2, resulting in faster access); Faster data access

Cons for inclusive: More time to cache; Cache may get filled quickly with data being replicated; Need to maintain correct state of data in both caches

Q5

5. Consider the following assembly-language program XYZ. The 'puts' function is a standard function declared in <stdio.h> with signature 'int puts(const char *s);'; it outputs its argument string to standard output with a newline appended, and returns a nonnegative integer on success and a negative integer on failure.

```
1    f:
2          subq    $8, %rsp
3          call    puts
4          xorl    %eax, %eax
5          addq    $8, %rsp
6          ret
7          .globl  main
8    main:
9          pushq   %rbx
10         leaq    8(%rsi), %rbx
11         subq    $16, %rsp
12         jmp     .L10
13   .L6:
14         xorl    %esi, %esi
15         addq    $8, %rbx
16         movl    $f, %edx
17         movq    %rsp, %rdi
18         call    pthread_create
19   .L10:
20         movq    (%rbx), %rcx
21         testq   %rcx, %rcx
22         jne     .L6
23         addq    $16, %rsp
24         xorl    %eax, %eax
25         popq    %rbx
26         ret
```

a. What would go wrong if we deleted lines 9 and 25?
b. What would go wrong if we deleted lines 2 and 5?
c. This machine code has a bug. What is it?

Answer 5

Q.5 a. The callee saved register RBX is being stomped on. %rbx value being altered.
Q.5 b. For Stack Alignment. Need memory for PUTS function. Partial credit if either of the 2 points missing.
Q.5 c. Call pthread_join with the appropriate arguments. Possible solution to resolve the bug would be having an array of thread IDs, initialized by the loop that calls pthread_create, and then used by a later loop that calls pthread_join. Partial marks for stating that the output string is being printed in random (maybe even interleaved) order (race condition). Partial credit also for stating that threads are not being cancelled or no exit.

Q5d

5d (11 minutes). Translate program XYZ to the equivalent C code. Your C program should have the same bug that the machine code does.

Answer 5d

```c
void *f(const char *s)
{
    puts(s);
}
int main(int argc, char *argv)
{
    pthread *x;
    char **p = &argv[1];
    while(*p)
    {
        pthread_create(&x,f,0,*p++);
    }
    return 0;
}
```

Synchronization

- Threads allow a lightweight way to perform concurrency with shared variables
  - "With great power, comes great responsibility…"

Concurrency bugs: Program *sometimes* works, data is *sometimes* wrong, crashes *sometimes*…

Must carefully govern access to shared variables!

Synchronization + Race Conditions

- One of the major advantages of using threads over processes is the ease by which threads can share data.
- This of course, is marred by the potential that by using threads, you get the wrong result.
- That's bad.
- How do we ensure that behavior is correct, even when the execution order is not guaranteed?

Synchronization + Race Conditions

- We can enforce protection for shared variables or critical sections with semaphores, which are basically locks that tell us how many threads can enter a section of code.
- The semaphore is of type: sem_t and it is sort of a glorified counter or more conceptually, a door that allows/prevents entry into a section of code.

Synchronization + Race Conditions

- While the counter is non-zero, the door is open and thread can pass. When this happens, the counter decrements.
- When the counter is zero, the door is closed. The thread must wait for the door to be open again (counter becomes non-zero) before it can pass.

Synchronization + Race Conditions

- sem_t sem;
- sem_wait(&sem) – If sem is non-zero, return and decrement sem (the door is open, thread is allowed to pass). If sem is zero, wait until sem is non-zero, then decrement and return (the door is closed, wait for it to open).
- sem_post(&sem) – Increment sem by one. If sem was previously zero, the door was closed. Open the door and allow another thread in.

Example: Bounded Shared Buffer

- "Producer/Consumer" Scenario
- Application: Playing a video
  - Video decoder is constantly decoding frames and placing them in a buffer (ie each frame is an image)
  - Video player is constantly taking images from the buffer, and displaying them on the screen
- Guard access to buffer carefully

Synchronization + Race Conditions

- Say we have the following line of code:
  - <line of code>
- ...and we want to allow only two threads to be able to execute that line of code at any time.
- sem_t sem;
- sem_init(&sem, 0, 2); //0 is an options argument, 2 is the number of threads allowed
- sem_wait(&sem);
- <line of code>;
- sem_post(&sem);

Synchronization + Race Conditions

- Let's say we have some shared resource that you can read and write from. This can be done via two functions:

```
void read()
{
  <read shared resource>
}

void write()
{
  <write shared resource>
}
```

Synchronization + Race Conditions

- However, we want the following conditions:
  - There can be an unbounded number of concurrent readers.
  - There can be only 1 writer and if someone is writing, there can be no readers.
- How can we go about this?

Synchronization + Race Conditions

```
void read()
{
 <read>
}
```

```
void write()
{
 <write>
}
```

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r);
  <read>
  sem_post(&r);
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- We'll definitely want some semaphores around the critical sections in read/write.
- Is this right?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r);
  <read>
  sem_post(&r);
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- This prevents multiple writers, but it allows writing to happen at the same time as reading.
- Also, what do we initialize r to We want unbounded readers.
- Let's try to solve the "writing/reading at the same time" problem first

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  sem_wait(&w)
  <read>
  sem_wait(&w)
  sem_wait(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Now, writing cannot happen at the same time as reading.
- What's the next step?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  sem_wait(&w)
  <read>
  sem_wait(&w)
  sem_wait(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- If we initialized r to, say 10, we could not have 10 readers reading at the same time because w is initialized to 1.
- We recognize that we only want to run sem_wait(&w) when there is atleast one reader.

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  sem_wait(&w)
  <read>
  sem_wait(&w)
  sem_wait(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- If no readers, there might be a writer. Therefore, a reader must call sem_post(&w) to allow writer.
- If there are readers, then that means there cannot be a writer.
- Therefore, the reader needs to call sem_wait(&w)

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
void read()
{
  sem_wait(&r)
  if(there is one reader)
    sem_wait(&w)

  <read>

  if(this is the last reader)
  sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Once a reader has finished reading, it must also check: if it is the last reader, then it should be able to allow a writer. Therefore it must release the lock by calling sem_post(&w)

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r)
  reader_count++;
  if(read_count == 1)
  sem_wait(&w)

  <read>

  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Does this work?

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r)
  reader_count++;
  if(read_count == 1)
  sem_wait(&w)

  <read>

  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- Two problems, first of all, this only works if we can set r to infinity, which we can't.
- Second, there's still a possible race condition. What if two reader threads increment reader_count before either can get to reader_count == 1? The whole system messes up.

Synchronization + Race Conditions

```
sem_t r; // init to ?
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r)
  reader_count++;
  if(read_count == 1)
  sem_wait(&w)

  <read>

  reader_count--;
  if(reader_count == 0)
    sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- That this treats the <read> as the critical section, but that's not what we want to protect.
- We want to make sure that infinite readers can read, but only one reader can increment reader_count at a time.

Synchronization + Race Conditions

```
sem_t r; // init to 1
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r);
  reader_count++;
  if(read_count == 1)
  sem_wait(&w);
  sem_post(&r);
  <read>
  sem_wait(&r);
  reader_count--;
  if(reader_count == 0)
  sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- It satisfies the constraints (single writer, multiple reader).
- Is there maybe still a problem though?

Synchronization + Race Conditions

```
sem_t r; // init to 1
sem_t w;// init to 1
int reader_count = 0;
void read()
{
  sem_wait(&r);
  reader_count++;
  if(read_count == 1)
  sem_wait(&w);
  sem_post(&r);
  <read>
  sem_wait(&r);
  reader_count--;
  if(reader_count == 0)
  sem_post(&w)
  sem_post(&r)
}
```

```
void write()
{
  sem_wait(&w);
  <write>
  sem_post(&w);
}
```

- This method is unfair to writers.
- That is to say, a potential writer could be starved since if readers keep coming in, the writer will never have a chance to write.

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i,
  *ptr;  for (i=0; i<N; i++) {
    ptr = Malloc(sizeof(int)); *ptr =
    i;
    Pthread_create(&tid[i],NULL,fn,ptr)
    ;
  }
  for (i=0; i<N; i++)
    Pthread_join(tid[i], NULL);
  exit(0);
}
```

```
void *fn(void *vargp) {
  int myid = *((int *)vargp);
  Free(vargp);
  printf("%d\n",myid);
  return NULL;
}
```

Question: Are there any race conditions in this code?

Answer: Nope. Careful use of Malloc/Free prevents possible bugs.

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i,
  *ptr;  for (i=0; i<N; i++) {
    ptr = Malloc(sizeof(int)); *ptr =
    i;
    Pthread_create(&tid[i],NULL,fn,ptr)
    ;
  }
  for (i=0; i<N; i++)
}   Pthread_join(tid[i], NULL);
  exit(0);
```

```
void *fn(void *vargp) {
  int myid = *((int *)vargp);
  Free(vargp);
  process(myid);
  return NULL;
}
```

Question: Outline an approach to avoid race conditions that doesn't use Malloc/Free. What are the advantages/disadvantages of your approach?

Q: Synchronization

```
int main() {
  pthread_t tid[N]; int i, *ptr;
  for (i=0; i<N; i++) {
    Pthread_create(&tid[i],NULL,fn,(void*)i)
    ;
  }
  for (i=0; i<N; i++)
    Pthread_join(tid[i], NULL);
} exit(0);
```

```
void *fn(void *vargp) {
  int myid = (int) vargp;
  process(myid);
  return NULL;
}
```

Answer: Simply pass in the int directly!
Pro: No added overhead due to malloc/free.
Con: Assumes that pointer datatype is at least bigger than size of int. May not be true on all systems.

Question

Consider the procedure below for the Producer-Consumer problem which uses semaphores:

```
semaphore n = 0;
semaphore s = 1;
void producer()
{
      while(true)
      {
       produce();
       semWait(s);
       addToBuffer();
       semSignal(s);
       semSignal(n);
      }
}
```

```
void consumer()
{
      while(true)
      {
       semWait(s);
       semWait(n);
       removeFromBuffer();
       semSignal(s);
       consume();
      }
}
```

Which of the following are true?

1. The producer will be able to add an item to the buffer, but the consumer can never consume it.
2. The consumer will remove no more than one item from the buffer.
3. Deadlock occurs if the consumer succeeds in acquiring semaphore s when the buffer is empty.
4. The starting value for the semaphore n must be 1 and not 0 for deadlock-free operation.

Answer

3 is true.
Initially, there is no element in the buffer.
Semaphore s = 1 and semaphore n = 0.
We assume that initially control goes to the consumer when buffer is empty.
semWait(s) decrements the value of semaphore ‘s’ . Now, s = 0 and semWait(n) decrements the value of semaphore ‘n’. Since, the value of semaphore ‘n’ becomes less than 0 , the control stucks in while loop of function semWait() and a deadlock arises.

Thus, deadlock occurs if the consumer succeeds in acquiring semaphore s when the buffer is empty.

Intro to Deadlocks

- 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.

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

- 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: 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

Q: Deadlock

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Question: What are the possible outputs of this program? Explain your answer.

Q: Deadlock

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Answer: Nothing - this program will always deadlock!

Q: Deadlock

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Question: Now, what are the possible outputs of the program? Can deadlock still happen?

Q: Deadlock

```
int main() {
  sem_t s, t;
  pthread_t tid1, tid2;
  int v1 = 1; int v2 = 2;
  Sem_init(&s, 0, 2);
  Sem_init(&t, 0, 2);
  P(&s); P(&t);
  Pthread_create(&tid1, NULL, fn, &v1);
  Pthread_create(&tid2, NULL, fn, &v2);
  while (1);
}
```

```
void* thread(void* vargp) {
  P(&s);
  V(&s);
  P(&t);
  V(&t);
  printf("HERE: %d\n",
      *((int*)vargp));
  return NULL;
}
```

Answer: Either "Here: 1" -> "Here: 2", or vice-versa. Dead lock can't happen anymore.

Q: Deadlock

```
Thread 1:
  P(&s)
  P(&t)
  do_work();
  V(&t)
  V(&s)
```

```
Thread 2:
  P(&t)
  P(&s)
  do_work();
  V(&s)
  V(&t);
```

```
sem_t t;     // N = 1
sem_t s;     // N = 1
```

Will this always deadlock? Sometimes deadlock? Never deadlock? Show execution order for possible cases.

Q: Deadlock

Thread 1:

```
P(&s)
P(&t)
do_work();
V(&t)
V(&s)
```

Thread 2:

```
P(&t)
P(&s)
do_work();
V(&s)
V(&t);
```

```
sem_t t;     // N = 1
sem_t s;     // N = 1
```

Deadlock:

```
T1        T2
P(&s)
          P(&t)
          P(&s)
P(&t)
  T1,T2 stuck!
```

OK:

```
T1        T2
P(&s)
P(&t)
do_work()
V(&t)
V(&s)
          P(&t)
          P(&s)
          ...
```

Q: Deadlock

```
Thread 1:
   P(&t)
   P(&s)
   do_work();
   V(&s)
```

```
Thread 2:
   P(&t)
   P(&s)
   do_work();
   V(&s)
   V(&t);
```

```
sem_t t;     // N = 1
sem_t s;     // N = 1
```

Will this always deadlock? Sometimes deadlock? Never deadlock? Show execution order for possible cases.

Q: Deadlock

Thread 1:
```
P(&t)
P(&s)
do_work();
V(&s)
```

Thread 2:
```
P(&t)
P(&s)
do_work();
V(&s)
V(&t);
```

```
sem_t t;     // N = 1
sem_t s;     // N = 1
```

OK:
```
T1        T2
        P(&t)
        P(&s)
        do_work()
        V(&s)
        V(&t)
P(&t)
...
```

Deadlock:
```
T1          T2
P(&t)
P(&s)
do_work()
V(&s)
            P(&t)
      T2 is stuck!
```

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-safe functions

Typically achieve thread-safety by mechanisms:

- Synchronization (ie semaphores/mutexes)
- Careful handling of shared data

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

- 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 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

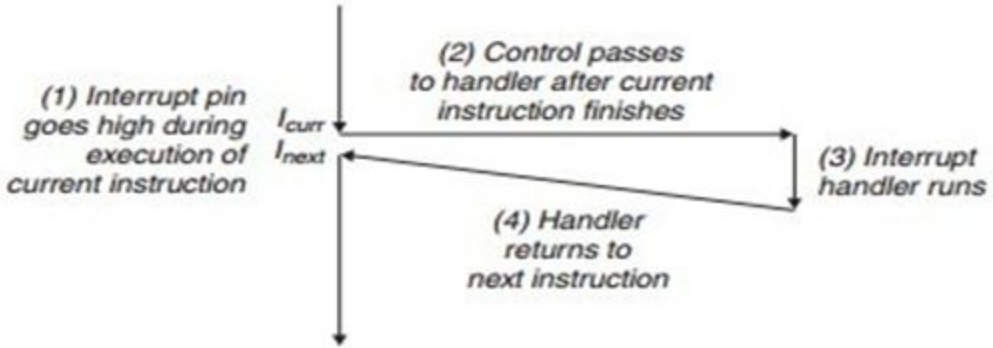
- Sometimes when your program is running, it has to be able to respond to outside stimuli.
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
- These signals will be sent to running programs.
- Asynchronous
  - Occurs independently of currently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



Reentrancy

- Thus, if a program is running a particular thread and that thread is in a function, it is possible that an interrupt causes the thread's execution to be switched to different code to handle the interrupt.
- If the interrupt handler calls the same function that you were in when you were interrupted, 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

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

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

Reentrancy

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

Reentrancy

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

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

Reentrancy

- …
  strcpy(ptr, local_ptr);
  local_var = 10; ← Current, ptr = “interrupt”
  return ptr; ← Interrupt Handler
  }

- When the interrupt handler completes, the thread goes back to the instruction that it would have executed before the interrupt.

Reentrancy

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

- Now the result is wrong.

Reentrancy

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

```
1   char *ctime_ts(const time_t *timep, char *privatep)
2   {
3       char *sharedp;
4
5       sem_wait(&mutex);
6       sharedp = ctime(timep);
7       strcpy(privatep, sharedp); /* Copy string from shared
to private */
8       sem_post(&mutex);
9       return privatep;
10  }
```

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 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

```
1   char *ctime_ts(const time_t *timep, char *privatep)
2   {
3       char *sharedp;
4
5       P(&mutex);  ← Current
6       sharedp = ctime(timep);
7       strcpy(privatep, sharedp);
8       V(&mutex);
9       return privatep;
10  }
```

Reentrancy

```
1   char *ctime_ts(const time_t *timep, char *privatep)
2   {
3       char *sharedp;
4
5       P(&mutex);
6       sharedp = ctime(timep); ← Current, mutex = 0
7       strcpy(privatep, sharedp);
8       V(&mutex);
9       return privatep;
10  }
```

- INTERRUPT!
- Interrupt calls ctime_ts.

Reentrancy

```
1   char *ctime_ts(const time_t *timep, char *privatep)
2   {
3       char *sharedp; ← Interrupt
4
5       P(&mutex);
6       sharedp = ctime(timep); ← Current, mutex = 0
7       strcpy(privatep, sharedp);
8       V(&mutex);
9       return privatep;
10  }
```

Reentrancy

```
1   char *ctime_ts(const time_t *timep, char *privatep)
2   {
3       char *sharedp;
4
5       P(&mutex);  ← Interrupt
6       sharedp = ctime(timep); ← Current, mutex = 0
7       strcpy(privatep, sharedp);
8       V(&mutex);
9       return privatep;
10  }
```

- When the interrupt 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.

Example: sum

```
int result = 0;
void sum_n(int n) {
  if (n == 0) {
    result = n;
  } else {
    sum_n(n-1);
    result = result + n;
  }
}
```

Suppose Kim tries to make this code thread safe...

Example: fib

Question: Is there anything wrong with this code?

```
int result = 0;
sem_t s; // sem_init(&s,1);
void sum_n(int n) {
  if (n == 0) {
    P(&s); result = n; V(&s);
  } else {
    P(&s);
    sum_n(n-1);
    result = result + n;
    V(&s);
  }
}
```

Answer: Yes, deadlock! sum_n(5) calls sum_n(4), but sum_n(4) can't acquire mutex. sum_n(5) can't make progress without sum_n(4) - thread is stuck.

Ex: strtoupper

```
/* non-reentrant function */
char *strtoupper(char *string) {
  static char buffer[MAX_STRING_SIZE];
  int index;
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Question: Is this threadsafe?

Answer: Nope! Two threads running strtoupper() will write to shared buffer.

Ex: strtoupper

```
/* reentrant function (a poor solution) */
char *strtoupper(char *string) {
  char *buffer;
  int index;
  /* error-checking should be performed! */
  buffer = malloc(MAX_STRING_SIZE);
  for (index = 0; string[index]; index++)
    buffer[index] = toupper(string[index]);
  buffer[index] = 0
  return buffer;
}
```

Ex: strtoupper

```
/* reentrant function (a better solution) */
char *strtoupper_r(char *in_str, char *out_str) {
  int index;
  for (index = 0; in_str[index]; index++)
    out_str[index] = toupper(in_str[index]);
  out_str[index] = 0
  return out_str;
}
```

Reentrancy vs Thread Safety

Question: Are threadsafe functions always reentrant?

```
void f() {
  mutex_acquire();
  // suppose signal handler gets invoked here!
  do_important_stuff();
  mutex_release();
}
```

Answer: Nope! Suppose function f() is used as a signal handler. Suppose we are executing f(), and acquire the mutex. Then, suppose signal handler gets invoked again, and we invoke f() again. The signal handler will get stuck trying to acquire the mutex!

Reentrancy vs Thread Safety

Question: Are reentrant functions always threadsafe?

Answer: According to your textbook, yes. This is using the definition that reentrant functions never access shared data.

References

- Slides modified from DJ Kim, UT Wang and Shikhar Malhotra
- http://www.cs.cmu.edu/afs/cs/academic/class/15213-f15/www/schedule.html

Thank You