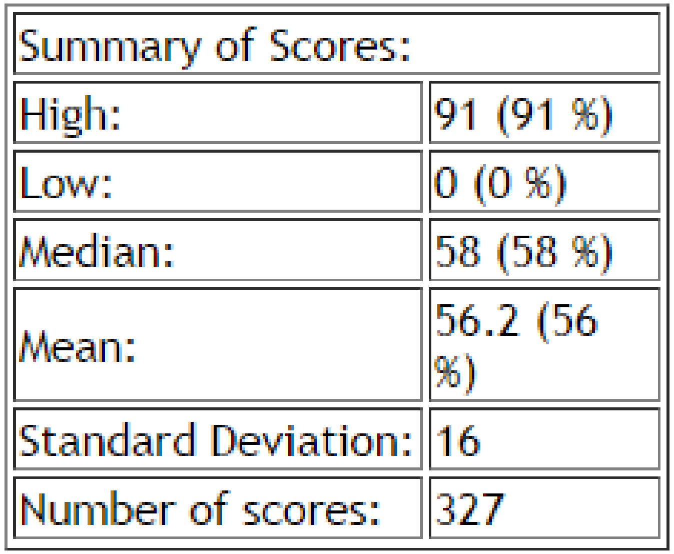
Introduction to Computer Organization

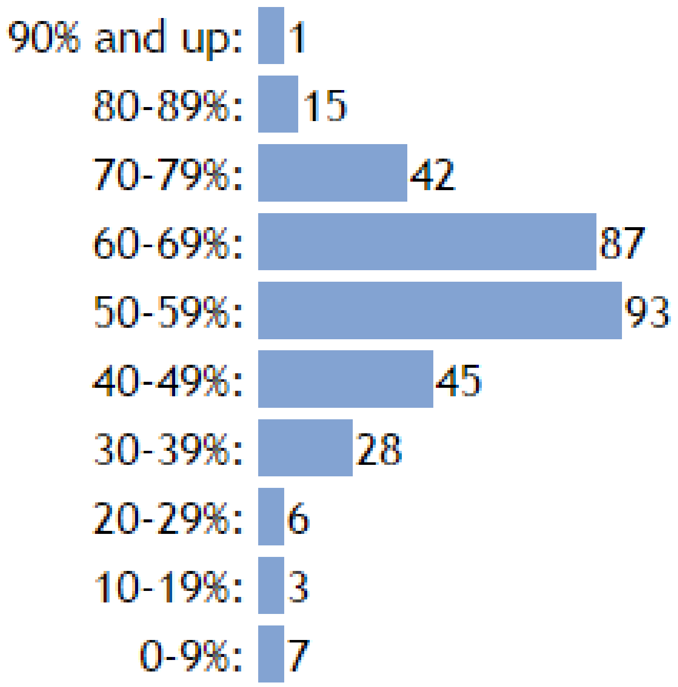
DIS 1H – Week 8

Slides modified from Eric Kim

Midterm Distribution



Summary of Scores:	
High:	91 (91 %)
Low:	0 (0 %)
Median:	58 (58 %)
Mean:	56.2 (56 %)
Standard Deviation:	16
Number of scores:	327



Agenda

- Synchronization
- Deadlocks
- Thread Safety and Reentrancy
- I/O

Synchronization

- Threads allow a lightweight way to perform concurrency with shared variables

  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);
}
```

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

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

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 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!
Adv: No added overhead due to malloc/free.

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

- According to the book, 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

- 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

- 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.

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

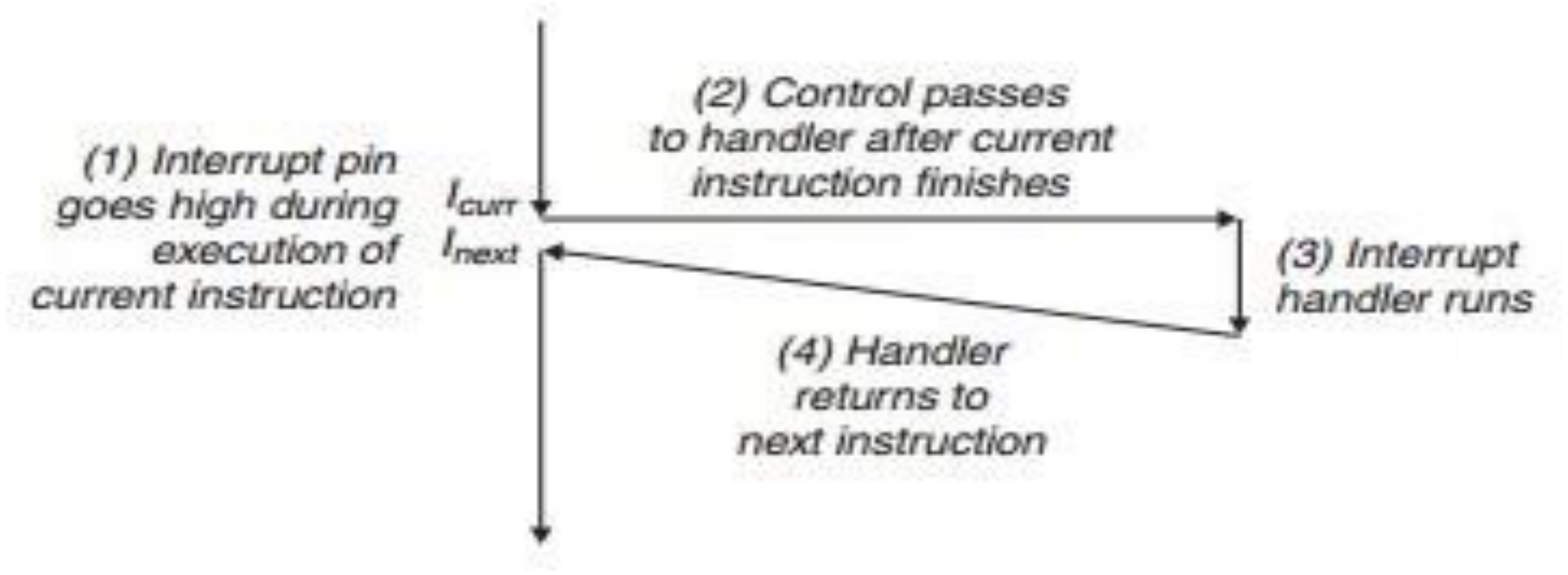
- Sometimes when your program is running, it has to be able to respond to outside stimuli.
- Most commonly signals from I/O devices.
  - Keyboard key presses.
  - Mouse movement
  - Network adapter activity
- These signals will be sent to running programs.
- Asynchronous
  - Occurs independently of currently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

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:

```
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 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;
}
```

- INTERRUPT!
- Interrupt calls ctime_ts.

Reentrancy

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

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

- 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.

Supplemental Readings

Helpful reading on threads:

https://randu.org/tutorials/threads/

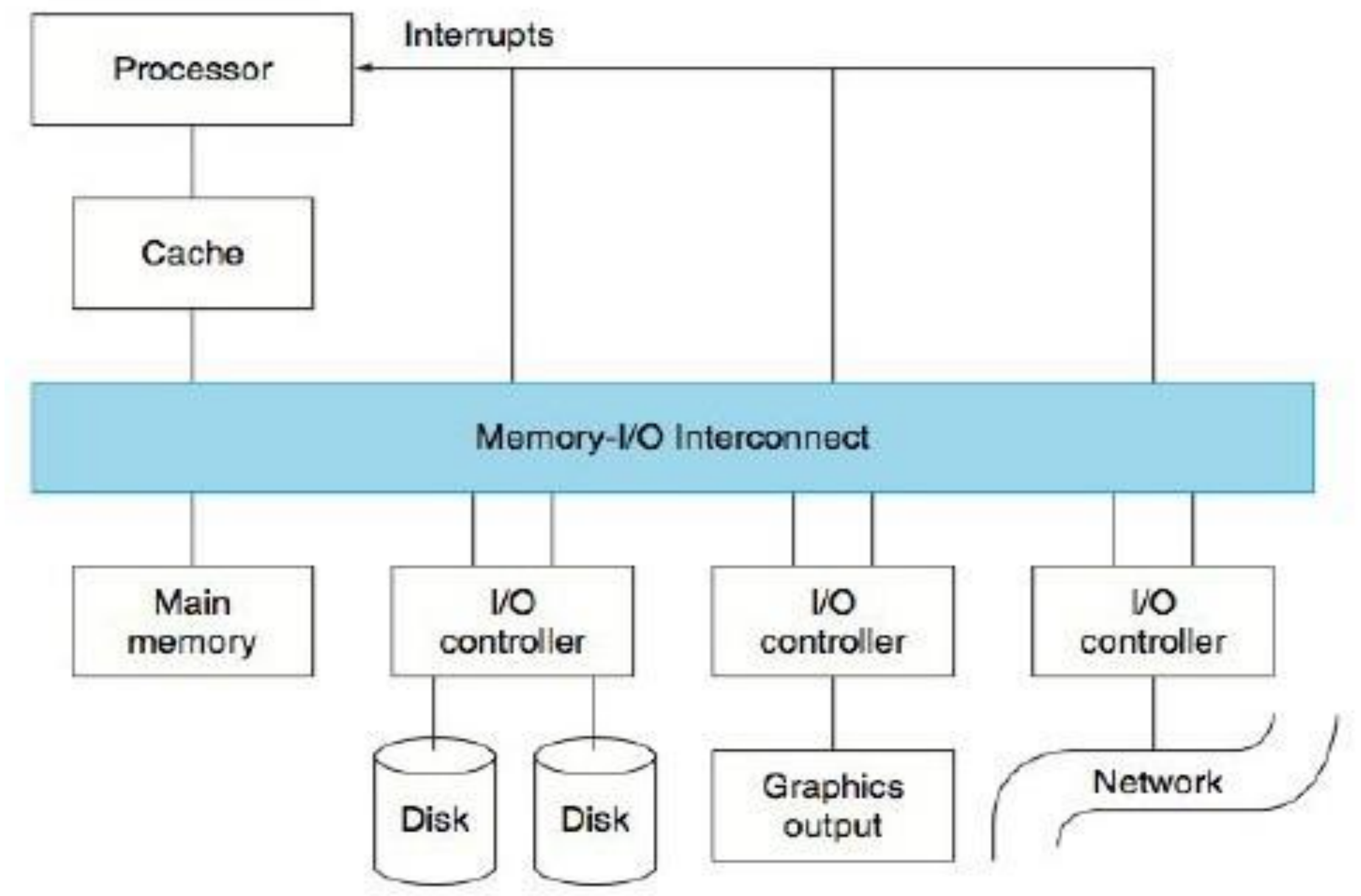
Helpful reading on thread safety and reentrancy:

https://www-01.ibm.com/support/knowledgecenter/ssw_aix_61/com.ibm.aix.genprogc/writing_reentrant_thread_safe_code.htm?cp=ssw_aix_61%2F13-3-12-18

.

The more hardware-ish viewpoint

- A hypothetical logical interconnection:



← Bus

I/O

- Things to note:
- Despite the variety of components (CPU, memory, devices), most things are linked together via a “bus”.
- A bus is essentially a collection of wires that can be used to transfer data.
- In this example the processor is not connected to the cache via the bus. Generally, we want at least the L1 cache to be as close to the processor as possible.

I/O

- Things to note:
- Devices are not directly connected to the bus. They are connected via an I/O controller.
- What are the trade-offs of having everything connected via a single bus?

I/O

- What are the trade-offs of having everything connected via a single bus?
- Pros:
  - Simple from a hardware standpoint. Requires few components/little space.
  - Transparency between devices. Each device can snoop on the bus and know exactly what's going on. This is a very nice property if you have multiple processors sharing memory via a bus.
- Cons?

I/O

- What are the trade-offs of having everything connected via a single bus?
- Cons:
  - You could potentially have many devices competing for a single resource. The bus becomes a huge bottleneck.
  - Bus arbitration logic. If a device wants to use a bus, it needs to fight the other devices for the right to use the bus. There is only one winner.

I/O

- Things to note:
- The wires of the bus fall into several categories and serve different purposes (data, address, control).
- Through the bus, each of the controllers can communicate with the other controllers (depending on the complexity of the controller)

I/O

- Computer architecture and interconnection is not standardized because different goals require different interconnections.
- There are, however, common features that you should be aware of so that you will be able to understand more practical/realistic arrangements.

Communication with I/O Devices

- At the processor, how are commands given to I/O devices?
- Memory Mapped I/O: Portions of the address space are assigned to represent an I/O device.
  - movl (0x1234), %eax
  - The address and operation are sent over the bus. The memory system recognizes that the address corresponds to an I/O device and doesn't respond.
- Special I/O assembly instructions.

Communication with I/O Devices

- Now that the processor has issued a request to a device, how does the I/O device respond back to the processor?
  - Polling
  - Interrupt driven I/O
  - Direct Memory Access

Communication with I/O Devices

- Polling
  - When the I/O device has a result ready for the CPU, it indicates it or puts the result in a register.
  - When the CPU expects a result, it must manually go and check the registers to see if it has a delivery. If there is, the CPU goes and gets the result.
  - If the CPU must wait before continuing, it can issue a request and do nothing other than continually poll the register until a result is received.
  - What about if a CPU isn't explicitly waiting for something (like a mouse/keyboard action)?

Communication with I/O Devices

- Polling
  - What about if a CPU isn't explicitly waiting for something (like a mouse/keyboard action)?
  - The CPU will have to spend a lot of time checking the status register to make sure there isn't some result that it has to handle.
  - This is a waste of time.

Communication with I/O Devices

- Interrupt Driven:
  - When an I/O device has a result. It will modify a status and cause register and inform the CPU by explicitly sending an interrupt.
  - Once the currently running instruction completes, the CPU will handle the exception and read the registers if necessary to decide how to handle the I/O.

Communication with I/O Devices

- Both interrupt-driven and polling simply inform the CPU of when it can fetch data and the CPU is ultimately responsible for getting the data.
- If the CPU is spending time fetching data from I/O devices, it's not running the program that it's supposed to be running. Program execution is halted.
- As a result, these methods work fine for small data transfers but don't work very well for large and slow data transfers (like reading from disk).

Communication with I/O Devices

- When the processor reads to disk, the processor will be copying from disk to RAM.
- What a bore.
- If only the CPU could spend it's time running the program and the “disk controller” could access memory... directly.

Communication with I/O Devices

- Direct Memory Access (DMA)
- Turns the I/O controllers into small CPUs, each of which can access the bus and do work such as transferring data.
    1. Processor issues request informing the DMA controller what to do.
    2. Processor goes back to doing it's job. Meanwhile the DMA controller arbitrates for the bus and attempts to do the data transfer to memory.
    3. When the DMA is done, it informs the CPU via an interrupt.

Communication with I/O Devices

- In this way, the processor can do the haughty processor work without having to sully itself with the petty matters of the proletariat (I/O devices).

What this means for you

- When it comes to applications, how can they make use of this I/O structure?

What this means for you

- Initiate the low level interactions by simply calling syscalls:
- open()
- read()
- write()
- close()