Final Review

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk
Angie Cao
Aniket Ghorpade
Ashwin Alaparthi
Austin Lin
Hassan Rizwan
Lobi Zhao
Pearl Liu
Perrie Quek

Administrivia

- Final Project Autograder Posted
  - SOME of it is auto graded. There is a lot of functionality that is not autograded that you will need to implement
  - Extended to Midnight on Thursday

- This lecture: Exam Review
- Travis will still have OH this Friday, most TA's have finished OH
- Re-opens should happen soon

- End of semester survey posted, due Tuesday the 6th

- Exam logistics & Practice exam questions posted! (Solutions over weekend!)

Exam Philosophy / Advice (pt. 1)

- I do not like midterms that ask you to memorize things
  - You will still have to memorize some critical things.
  - I will hint at some things, provide documentation or a summary of some things. (for example: I will list some of the functions that may be useful and a brief summary of what the function does)

- I am more interested in questions that ask you to:
  - Apply concepts to solve new problems
  - Analyze situations to see how concepts from lecture apply

- Will there be multiple choice?
  - If there is, you will still have to justify your choices

Exam Philosophy / Advice (pt. 2)

- I am still trying to keep the exam fair to you, you must remember some things
  - High level concepts or fundamentals. I do not expect you to remember every minute detail.
    - E.g. how a multi level page table works should be know, but not the exact details of what is in each page table entry
    - (I know this boundary is blurry, but hopefully this statement helps)

- I am NOT trying to “trick” you (like I sometimes do in poll everywhere questions)

Exam Philosophy / Advice (pt. 3)

- I am trying to make sure you have adequate time to stop and think about the questions.
  - You should still be wary of how much time you have
  - But also, remember that sometimes you can stop and take a deep breath.

- Remember that you can move on to another problem.

- Remember that you can still move on to the next part even if you haven't finished the current part

Exam Philosophy / Advice (pt. 4)

- On the midterm you will have to explain things
- Your explanations should be more than just stating a topic name.
- Don't just say something like (for example) "because of threads" or just state some facts like "threads are parallel and lightweight processes".
- State how the topic(s) relate to the exam problem and answer the question being asked.

Disclaimer

- **THIS REVIEW IS NOT EXHAUSTIVE**
- **Topics not in this review are still testable**
  - **We recommend going through the course material. Lecture polls, recitation worksheets, and the previous homework assignments.**

Review Topics

- C++ Programming
  - (Not included in this lecture, see the practice problems posted with exam policies)
- C++ Memory Diagram & Allocations
- C++ Copying
- Locality
- Inter Process Communication
- Process Synchronization
- Threads & Deadlocks
- Threads & Condition Variables
- Networking

C++ Memory Diagram & Allocatio

- Consider the following code that uses std::list (linked list)
- How many memory allocations occur in this code?
- What is the state of memory when we reach HERE?

```
int main() {
  list<coord> l;
  coord rn = {1, 1};
  l.push_back(rn);
  rn = {2, 2};
  l.push_back(rn);
  rn = {3, 3};
  l.push_back(rn);
  list<coord> result = std::move(scale(l));
  // HERE
}
```

```
struct coord {
  int x;
  int y;
}

list<coord> scale(list<coord> to_norm) {
  int total_x = 0;
  int total_y = 0;
  for (coord r : to_norm) {
    total_x += r.x;
    total_y += r.y;
  }

  for (coord& r : to_norm) {
    r.x *= total_x;
    r.y *= total_y;
  }

  return to_norm;  // result is moved
}
```

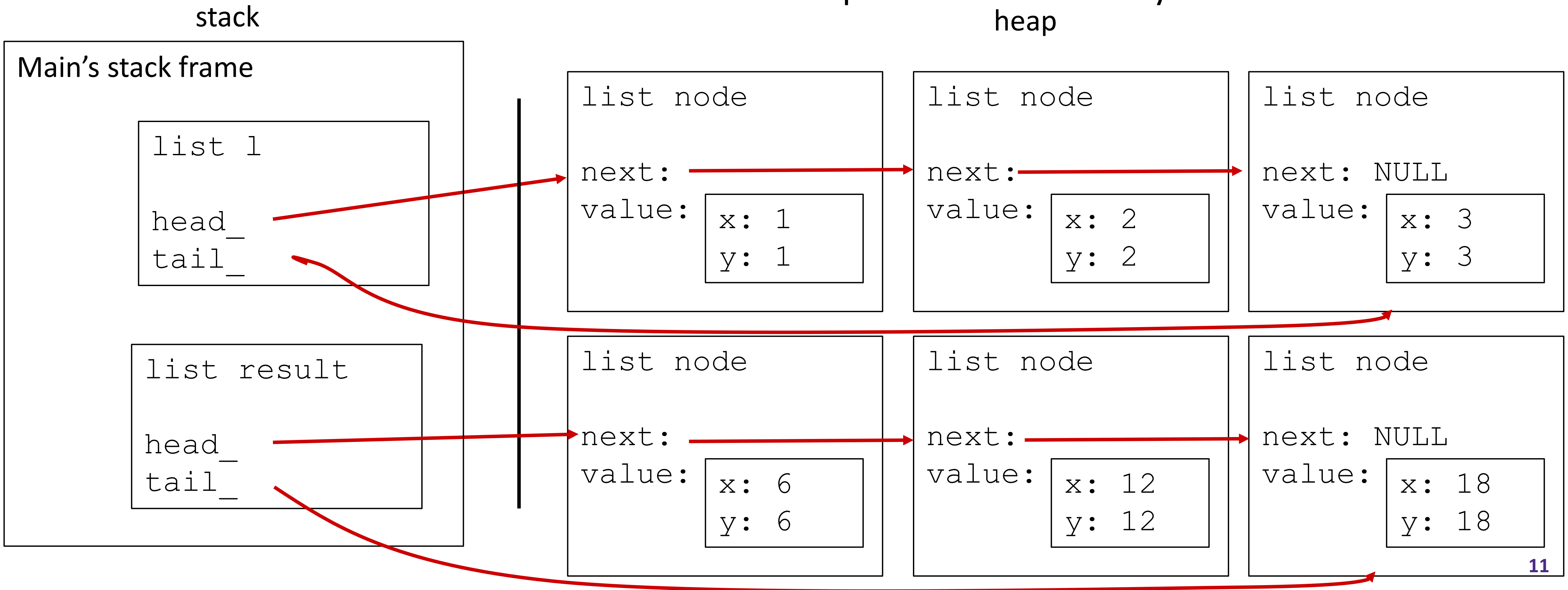
C++ Memory Diagram & Allocations

- Consider the following code that uses std::list (linked list)
- How many memory allocations occur in this code?
  - **0 for initial construction of the list in main**
  - **3 for push_back in main (1 for each node that must be allocated for the list)**
  - **3 for copy constructing the list as a parameter to scale()**
  - **0 for iterating ove the list in scale(). Yes we do make a copy of the coord structs, but those are just ints, no memory allocation needed**
  - **0 for moving the returned list out to the list result in main**

- **6 in total**

C++ Memory Diagram & Allocations

- Memory Diagram:
  - Since we didn't go over the exact internals of the linkedlist, it would have been fine to have a slightly different linked list structure (e.g. no tail_ pointer) as long as it was clear it was a linked list and the nodes were on the heap similar to how they are here:



C++ Copying

- Below is a class that represents a Multiple Choice answer

```
class MC {
 public:
  MC() : resp_(' ') { }
  MC(char resp) : resp_(resp) { }
  char get_resp() const { return resp_; }
  bool Compare(MC mc) const;
 private:
  char resp_;
}; // class MC
```

```
int QS 2
// this works
MC key[2] = {'D', 'A'};

size_t Score(const MC *ans) {
  size_t score = 0;
  for (int i = 0; i < QS; i++) {
    if (ans->Compare(key[i])) {
      score++;
    }
    ans++;
  }
  return score;
}
int main(int argc, char **argv) {
  MC myAns[QS];
  myAns[0] = MC('B');
  myAns[1] = MC('A');
  cout << "Score: ";
  cout << Score(myAns) << endl;
  return 0;
}
```

- How many times are each of the following invoked:
  - MC constructor
  - MC copy constructor
  - MC operator=
  - MC destructor

C++ Copying

- Below is a class that represents a Multiple Choice answer

```
class MC {
 public:
  MC() : resp_(' ') { }
  MC(char resp) : resp_(resp) { }
  char get_resp() const { return resp_; }
  bool Compare(MC mc) const;
 private:
  char resp_;
}; // class MC
```

```
int QS 2
// this works
MC key[2] = {'D', 'A'}; // ctor x2

size_t Score(const MC *ans) {
  size_t score = 0;
  for (int i = 0; i < QS; i++) {
    if (ans->Compare(key[i])) {
      score++;
    }
    ans++;
  }
  return score;
}
int main(int argc, char **argv) {
  MC myAns[QS];
  myAns[0] = MC('B');
  myAns[1] = MC('A');
  cout << "Score: ";
  cout << Score(myAns) << endl;
  return 0;
}
```

- How many times are each of the following invoked:
  - MC constructor
  - MC copy constructor
  - MC operator=
  - MC destructor

C++ Copying

- Below is a class that represents a Multiple Choice answer

```
class MC {
 public:
  MC() : resp_(' ') { }
  MC(char resp) : resp_(resp) { }
  char get_resp() const { return resp_; }
  bool Compare(MC mc) const;
 private:
  char resp_;
}; // class MC
```

- How many times are each of the following invoked:
  - MC constructor
  - MC copy constructor
  - MC operator=
  - MC destructor

```
int QS 2
// this works
MC key[2] = {'D', 'A'}; // ctor x2

size_t Score(const MC *ans) {
  size_t score = 0;
  for (int i = 0; i < QS; i++) {
    if (ans->Compare(key[i])) {
      score++;
    }
    ans++;
  }
  return score;
}
int main(int argc, char **argv) {
  MC myAns[QS];  // defulat ctor x2
  myAns[0] = MC('B');
  myAns[1] = MC('A');
  cout << "Score: ";
  cout << Score(myAns) << endl;
  return 0;
}
```

C++ Copying

- Below is a class that represents a Multiple Choice answer

```
class MC {
 public:
  MC() : resp_(' ') { }
  MC(char resp) : resp_(resp) { }
  char get_resp() const { return resp_; }
  bool Compare(MC mc) const;
 private:
  char resp_;
}; // class MC
```

- How many times are each of the following invoked:
  - MC constructor
  - MC copy constructor
  - MC operator=
  - MC destructor

```
int QS 2
// this works
MC key[2] = {'D', 'A'}; // ctor x2

size_t Score(const MC *ans) {
  size_t score = 0;
  for (int i = 0; i < QS; i++) {
    if (ans->Compare(key[i])) {
      score++;
    } // cctor in loop 2x for param
    ans++;
  }
  return score;
}
int main(int argc, char **argv) {
  MC myAns[QS];  // defulat ctor x2
  myAns[0] = MC('B'); // ctor then =
  myAns[1] = MC('A'); // ctor then =
  cout << "Score: ";
  cout << Score(myAns) << endl;
  return 0;
}
```

C++ Copying

- Below is a class that represents a Multiple Choice answer

```
class MC {
 public:
  MC() : resp_(' ') { }
  MC(char resp) : resp_(resp) { }
  char get_resp() const { return resp_; }
  bool Compare(MC mc) const;
 private:
  char resp_;
}; // class MC
```

```
int QS 2
// this works
MC key[2] = {'D', 'A'}; // ctor x2

size_t Score(const MC *ans) {
  size_t score = 0;
  for (int i = 0; i < QS; i++) {
    if (ans->Compare(key[i])) {
      score++;
    }
    ans++;
  }
  return score;
}
int main(int argc, char **argv) {
  MC myAns[QS];  // defulat ctor x2
  myAns[0] = MC('B'); // ctor then =
  myAns[1] = MC('A'); // ctor then =
  cout << "Score: ";
  cout << Score(myAns) << endl;
  return 0;
}
```

- How many times are each of the following invoked:
  - MC constructor
  - MC copy constructor
  - MC operator=
  - MC destructor

C++ Copying

- Below is a class that represents a Multiple Choice answer

```
class MC {
 public:
  MC() : resp_(' ') { }
  MC(char resp) : resp_(resp) { }
  char get_resp() const { return resp_; }
  bool Compare(MC mc) const;
 private:
  char resp_;
}; // class MC
```

```
int QS 2
// this works
MC key[2] = {'D', 'A'}; // ctor x2

size_t Score(const MC *ans) {
  size_t score = 0;
  for (int i = 0; i < QS; i++) {
    if (ans->Compare(key[i])) {
      score++;
    } // cctor in loop 2x for param
    ans++;
  }
  return score;
}
int main(int argc, char **argv) {
  MC myAns[QS];  // defulat ctor x2
  myAns[0] = MC('B'); // ctor then =
  myAns[1] = MC('A'); // ctor then =
  cout << "Score: ";
  cout << Score(myAns) << endl;
  return 0;
}
```

- How many times are each of the following invoked:
  - MC constructor
  - MC copy constructor
  - MC operator=
  - MC destructor

C++ Copying

- Below is a class that represents a Multiple Choice answer

```
class MC {
 public:
  MC() : resp_(' ') { }
  MC(char resp) : resp_(resp) { }
  char get_resp() const { return resp_; }
  bool Compare(MC mc) const;
 private:
  char resp_;
}; // class MC
```

```
int QS 2
// this works
MC key[2] = {'D', 'A'}; // ctor x2

size_t Score(const MC *ans) {
  size_t score = 0;
  for (int i = 0; i < QS; i++) {
    if (ans->Compare(key[i])) {
      score++;
    } // cctor in loop 2x for param
    ans++;
  }
  return score;
}
int main(int argc, char **argv) {
  MC myAns[QS];  // defulat ctor x2
  myAns[0] = MC('B'); // ctor then =
  myAns[1] = MC('A'); // ctor then =
  cout << "Score: ";
  cout << Score(myAns) << endl;
  return 0;
}
```

- How many times are each of the following invoked:
  - MC constructor 6
  - MC copy constructor 2
  - MC operator= 2
  - MC destructor 8

Locality

- Typically, a `bool` variable is 1 byte. How much space does a `bool` strictly ***need*** though?
  - 1 bit

- C++ goes against the standard implementation of a vector for the bool type, and instead has each bool stored as a bit instead of the type a stand-a-lone Boolean variable would be stored as.
  - Travis thinks this was a horrible design decision, but there is a reason why they did this. What are those reasons?

Locality

- Typically, a `bool` variable is 1 byte. How much space does a `bool` strictly ***need*** though?
  - 1 bit

- C++ goes against the standard implementation of a vector for the bool type, and instead has each bool stored as a bit instead of the type a stand-a-lone Boolean variable would be stored as.
  - Travis thinks this was a horrible design decision, but there is a reason why they did this. What are those reasons?
  - **A lot less space is taken up, and as a side effect of that, you probably don't have to call malloc as often and will have better cache performance**

Locality

- If we stored a vector of 120 **`bool`**s, and wanted to iterate over all of them, roughly how many cache hits & misses would we have if we:
  - You can assume a cache line is 64 bytes.
  - If we used a **`vector<bool>`** that allocates the bools normally (1 byte per bool)
  - If we use a **`vector<bool>`** that represents each bool with a single bit

Locality

- If we stored a vector of 120 **`bool`**s, and wanted to iterate over all of them, roughly how many cache hits & misses would we have if we:
  - You can assume a cache line is 64 bytes.
  - If we used a **`vector<bool>`** that allocates the bools normally (1 byte per bool)
    - 2 cache misses, 118 cache hits
  - If we use a **`vector<bool>`** that represents each bool with a single bit
    - 1 cache miss, 119 cache hits

IPC

- The following code intends to use a global variable so that a child process reads a string and the parent prints it.

- Briefly describe two reasons why this program won't work. You can assume it compiles.

```
string message;

void child();
void parent();

int main() {
  pid_t pid = fork();
  if (pid == 0) {
    child();
  } else {
    parent();
  }
}
void child() {
  cin >> message;
}
void parent() {
  cout << message;
}
```

IPC

- The following code intends to use a global variable so that a child process reads a string and the parent prints it.

- Briefly describe two reasons why this program won't work. You can assume it compiles.
  - After fork is called, global variables are no longer shared. Each process has its own "message"
  - There is no synchronization to know if the parent prints after the child reads.

```
string message;

void child();
void parent();

int main() {
  pid_t pid = fork();
  if (pid == 0) {
    child();
  } else {
    parent();
  }
}
void child() {
  cin >> message;
}
void parent() {
  cout << message;
}
```

IPC

- Describe how we would have to rewrite the code if we wanted it to work. Keeping the multiple processes and calls to fork(). Be specific about where you would add the new lines of code.

```
string message;

void child();
void parent();

int main() {
  pid_t pid = fork();
  if (pid == 0) {
    child();
  } else {
    parent();
  }
}
void child() {
  cin >> message;
}
void parent() {
  cout << message;
}
```

IPC

- Describe how we would have to rewrite the code if we wanted it to work. Keeping the multiple processes and calls to fork(). Be specific about where you would add the new lines of code.

- ONE POSSIBLE ANSWER:

```
string message;
int fds[2];

void child();
void parent();

int main() {
  pipe(fds);
  pid_t pid = fork();
  if (pid == 0) {
    close(fds[0]);
    child();
  } else {
    close(fds[1]);
    parent();
  }
}
void child() {
  cin >> message;
  wrapped_write(fds[1], message);
}
void parent() {
  wrapped_read(fds[0], message);
  cout << message;
}
```

Process Synchronization

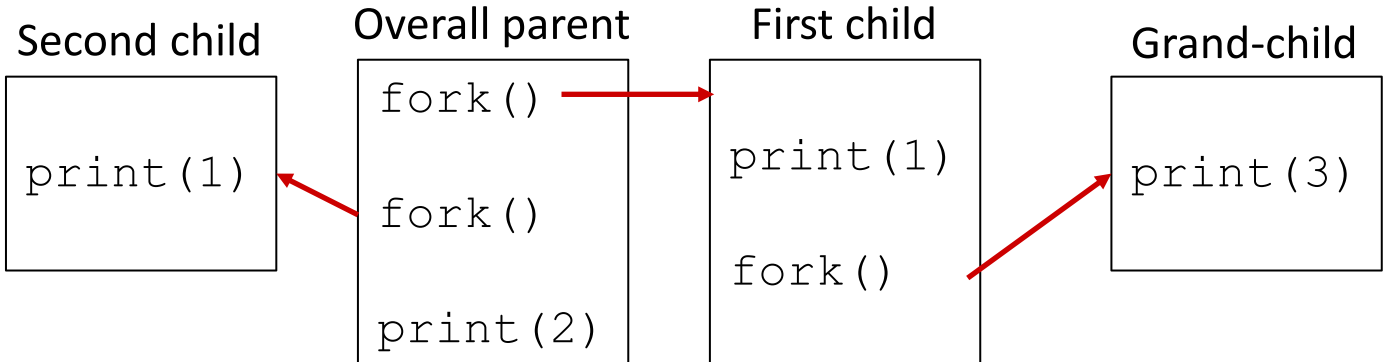
- Which of the following outputs are possible? How?
  - 1213
  - 3112
  - 2312
  - 1123
- If we wanted to change the code to guarantee that 1312 is printed. How could we do that?
  - There must still be 4 processes forked in a similar way (The initial process can't fork 3 direct children)
  - Each process must print out the same number as before.

```
int main() {
  pid_t pid = fork();
  bool flag = false
  if (pid == 0) {
    flag = true;
    cout << "1" << endl;
  }

  pid = fork();

  if (pid == 0) {
    if (flag) {
      cout << "3" << endl;
    } else {
      cout << "1" << endl;
    }
  } else if (!flag) {
    cout << "2" << endl;
  }
}
```

Process Synchronization

- Which of the following outputs are possible? How?
  - 1213 Possible
  - 3112 Not Possible
  - 2312 Not Possible
  - 1123 Possible

- If we draw the processes and ordering within a process we get:
  - Within each process the events must happen in that order. So the first child must print(1) before it forks the child that prints 3, so there must be a 1 printed before 3 is printed.



Process Synchronization

- If we wanted to change the code to guarantee that 1312 is printed. How could we do that?
  - There must still be 4 processes forked in a similar way (The initial process can't fork 3 direct children)
  - Each process must print out the same number as before.
  - One possible answer:
    - Main thing: using waitpid to enforce ordering
    - Make children processes exit to make sure it doesn't continue running code it shouldn't run.

```
int main() {
  pid_t pid = fork();
  bool flag = false
  if (pid == 0) {
    flag = true;
    cout << "1" << endl;
    pid = fork();
    if (pid == 0) {
      cout << "3" << endl;
      exit(EXIT_SUCCESS);
    }
    waitpid(pid, NULL, 0);
    exit(EXIT_SUCCESS);
  }
  waitpid(pid, NULL, 0);
  pid = fork();

  if (pid == 0) {
    cout << "1" << endl;
    exit(EXIT_SUCCESS);
  }
  waitpid(pid, NULL, 0);
  cout << "2" << endl;
}
```

Threads & Locks

- This code has the possibility to deadlock. Give an example of this happening. You can assume no thread tries to acquire the same lock twice
- Someone proposes we fix this by locking the whole database instead of locking at the block level. What downsides does this have? Does it even avoid deadlocks?
- How can we fix this (without locking the whole database if that even works)?

```
void transaction(list<int> block_numbers) {
  for (every block_num in block_numbers) {
    acquire_lock(block_num)
  }

  operation(block_numbers);

  for (every block_num in block_numbers) {
    release_lock(block_num);
  }
}
```

Threads & Locks

- This code has the possibility to deadlock. Give an example of this happening. You can assume no thread tries to acquire the same lock twice
  - **Thread 1 wants B2 and B4. Thread 2 also wants B2 and B4, but lists them in a different order. Thread 1 gets B2, Thread 2 get B4, and we deadlock.**

```
void transaction(list<int> block_numbers) {
  for (every block_num in block_numbers) {
    acquire_lock(block_num)
  }

  operation(block_numbers);

  for (every block_num in block_numbers) {
    release_lock(block_num);
  }
}
```

Threads & Locks

- Someone proposes we fix this by locking the whole database instead of locking at the block level. What downsides does this have? Does it even avoid deadlocks?
  - **This works, but now our data base is run entirely sequentially for these transactions even if two thread have completely separate blocks they operate on, they cannot run in parallel.**

```
void transaction(list<int> block_numbers) {
  for (every block_num in block_numbers) {
    acquire_lock(block_num)
  }

  operation(block_numbers);

  for (every block_num in block_numbers) {
    release_lock(block_num);
  }
}
```

Threads & Locks

- How can we fix this (without locking the whole database if that even works)?
- **Have each thread acquire the locks in a strict increasing numerical order. This prevents any cycles from happening**

```
void transaction(list<int> block_numbers) {
  for (every block_num in block_numbers) {
    acquire_lock(block_num)
  }

  operation(block_numbers);

  for (every block_num in block_numbers) {
    release_lock(block_num);
  }
}
```

Threads & Condition Variables

- If we have 7 threads all reading shared memory but not writing, is a data race possible?

- what if one of the 7 threads writes to the shared memory?

Threads & Condition Variables

- If we have 7 threads all reading shared memory but not writing, is a data race possible?
  - **No, a data race requires concurrent access of a shared resource and at least one thread is writing to that resource.**

- what if one of the 7 threads writes to the shared memory?
  - **Yes, it is possible. See definition above.**

Threads & Condition Variables

- We create these two functions for threads to read and write some shared memory but allows multiple readers. Something is wrong though, what is it? How do we fix?

```
void do_read() {
  pthread_mutex_lock(&lock);
  num_readers += 1;
  pthread_mutex_unlock(&lock);

  // don't hold the lock while reading
  // so that other readers can get access

  // do read (omitted for space)

  pthread_mutex_lock(&lock);
  num_readers -= 1;
  pthread_mutex_unlock(&lock);
}
```

```
int num_readers:
pthread_mutex_t lock
pthread_cond_t cond;

void do_write() {
  pthread_mutex_lock(&lock);
  while (num_readers > 0) {
    pthread_cond_wait(&cond, &lock);
  }

  // do write
  // (omitted for space)

  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&lock);
}
```

Threads & Condition Variables

- We create these two functions for threads to read and write some shared memory but allows multiple readers. Something is wrong though, what is it? How do we fix?

- **Do_read needs to signal or broadcast the condition variable after it decrements num_readers so that any waiting writer can wake up and check to see if it is ready for them to run.**

```
void do_read() {
  pthread_mutex_lock(&lock);
  num_readers += 1;
  pthread_mutex_unlock(&lock);

  // don't hold the lock while reading
  // so that other readers can get access

  // do read (omitted for space)

  pthread_mutex_lock(&lock);
  num_readers -= 1;
  if (num_readers == 0) {
    pthread_cond_signal(&cond);
  }
  pthread_mutex_unlock(&lock);
}
```

Networking: pt. 1 (True / False)

- TCP guarantees reliable delivery of the packets that make up a stream, assuming that the socket doesn't fail because of an I/O error.
- IP guarantees reliable delivery of packets, assuming that the socket doesn't fail because of an I/O error.
- Given a particular hostname (like www.amazon.com), getaddrinfo() will return a single IP address corresponding to that name.
- A single server machine can handle connection requests sent to multiple IP addresses.
- A struct sockaddr_in6 contains only an ipv6 address.
- The HTTP payload takes up a larger percentage of the overall packet sent over the network than the IP payload.

Networking: pt. 1 (True / False)

- TCP guarantees reliable delivery of the packets that make up a stream, assuming that the socket doesn't fail because of an I/O error.
  - True
- IP guarantees reliable delivery of packets, assuming that the socket doesn't fail because of an I/O error.
  - False
- Given a particular hostname (like www.amazon.com), getaddrinfo() will return a single IP address corresponding to that name.
  - False
- A single server machine can handle connection requests sent to multiple IP addresses.
  - True
- A struct sockaddr_in6 contains only an ipv6 address.
  - False
- The HTTP payload takes up a larger percentage of the overall packet sent over the network than the IP payload.
  - False

Networking pt. 2 (The one most reflective of an Exam Q)

- Pearl is setting up a C++ program to do network communication using UDP to send data.
  - She notices that when using UDP it is sometimes unreliable, and her packets do not get to there destinations in order. Is this a bug in how she wrote her program?
  - To try and remedy this issue, Pearl has each message she send contain a “Packet Number”. The receiver can then re-order the messages as they arrive to maintain the same order as sent. The receiver then sends messages back to acknowledge which packets it has received. Any messages not acknowledged are resent by the sender. Can this be implemented? What affect would it have on the order & reliability?

Networking pt. 2 (The one most reflective of an Exam Q)

- Pearl is setting up a C++ program to do network communication using UDP to send data.
  - She notices that when using UDP it is sometimes unreliable, and her packets do not get to there destinations in order. Is this a bug in how she wrote her program?
    - No, that's just UDP. UDP is the reason that packets are lost or show up in a different order. There isn't a way to mitigate this without doing extra stuff (like below). You could theoretically call this a bug by saying her choice to use UDP is a bug.

Networking pt. 2 (The one most reflective of an Exam Q)

- To try and remedy this issue, Pearl has each message she send contain a “Packet Number”. The receiver can then re-order the messages as they arrive to maintain the same order as sent. The receiver then sends messages back to acknowledge which packets it has received. Any messages not acknowledged are resent by the sender. Can this be implemented? What affect would it have on the order & reliability?
  - Yes, should help some. This is similar to what TCP does to ensure that data shows up in order.
  - This also is implementable. When we send data using UDP we just send bytes and specify how many bytes with sendto.

```
struct example_stuff_to_send {
  char characters[100];
};

example_stuff_to_send msg;
sendto(socket, &msg, sizeof(msg), addr, addr_len);
```

  - We can modify our struct to send additional data and if we are writing the code on the recievers end we can tell them to look at that message number and do the behavior specified.

```
struct example_stuff_to_send {
  int message_number;
  char characters[100];
};
```

Networking pt. 3

- For each of the following behaviors, identify what networking layer is most closely thought of as being responsible for handling that behavior.
  - Host A tries to send a long message to Host B in another city, broken up into many packets. A packet in the middle does not arrive, so Host A sends it again.
  - Host A tries to send a message to Host B, but Host C and Host D are also trying to communicate on the same network, so Host A must avoid interfering

Networking pt. 3

- For each of the following behaviors, identify what networking layer is most closely thought of as being responsible for handling that behavior.
  - Host A tries to send a long message to Host B in another city, broken up into many packets. A packet in the middle does not arrive, so Host A sends it again.
    - Transport Layer (Protocol commonly associated with this: TCP)
  - Host A tries to send a message to Host B, but Host C and Host D are also trying to communicate on the same network, so Host A must avoid interfering
    - Data Link Layer (Protocol commonly associated with this: MAC)

Networking pt. 4

- The original versions of HTTP (including 1.1) were designed to use plain text characters sent over the network instead of alternatives like a binary encoding for the request and response. Describe one advantage of this design decision and one disadvantage.

- Advantage:

- Disadvantage:

Networking pt. 4

- The original versions of HTTP (including 1.1) were designed to use plain text characters sent over the network instead of alternatives like a binary encoding for the request and response. Describe one advantage of this design decision and one disadvantage.

- Advantage:
  - Interpretable by humans
  - Easy to experiment with and adopt
- Disadvantage:
  - Might be less efficient (for some definition of efficient) than a well-packed binary format