Midterm Review

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat	Jialin Cai
Mati Davis	Donglun He
Chandravaran Kunjeti	Heyi Liu
Shufan Liu	Eddy Yang

Poll Everywhere

pollev.com/tqm

- Which topic do you want to cover for the review?

Upcoming Due Dates

- HW2 (Threads) Due TONIGHT @ 11:59 pm
  - Released

- Midterm
  - Take-home style on Wednesday 3/1 @ Noon till Friday 3/3 @ noon
  - Logistics released on Course Website

Collaboration Policy

- You are to write up and work on the midterm on your own. We want the work you submit to be a representation of your own thoughts.

- However, we acknowledge that your peers are often one of the best resources for understanding concepts; therefore, we are allowing the "Gilligan's Island Rule."

Gilligan’s Island Rule

- **The Gilligan's Island Rule:** You are free to meet with fellow students and discuss assignments with them. Writing on a board or shared piece of paper during the meeting is acceptable; however, **you should not take any written (electronic or otherwise) record away from the meeting.** Everything that you derive from the collaboration should be in your head.
- After the meeting, engage in at least a half-hour of mind-numbing activity (like watching an episode of Gilligan's Island), before starting to work on the assignment.

Review Topics

- C strings & output params
- C++ Class
- Concurrency
- Scheduling
- VM
- Caching (LRU)

C Strings & Output Params

- Complete the following function (on Codio)

```
// given a string, allocates and creates a new duplicate
// of it and returns it through the output parameter "out".
// Returns false on error, returns true otherwise

void str_duplicate(char* str, char** out) {









}
```

C Strings & Output Params

NOTE: There are other possible solutions

- Complete the following function

```c
// given a string, allocates and creates a new duplicate
// of it and returns it through the output parameter "out".
// Returns false on error, returns true otherwise

void str_duplicate(char* str, char** out) {
  char* res = malloc((strlen(str) + 1) * sizeof(char));
  *out = res;
  while(*str) {
    *res = *str
    str++;
    res++;
  }
  *res = *str;  // null terminator
  return true;
}
```

C Strings & Output Params

- Complete the main function

```
// given a string, duplicates it and returns it through
// the output parameter "out". Returns false on error
// returns true otherwise
void str_duplicate(char* str, char** out);

// duplicates a string literal,
// prints the duplicate, and runs without errors
int main(int argc, char** argv) {
  char* sample = "Hello World!";









}
```

C Strings & Output Params

NOTE: There are other possible solutions

- Complete the main function

```
// given a string, duplicates it and returns it through
// the output parameter "out". Returns false on error
// returns true otherwise
void str_duplicate(char* str, char** out);

// duplicates a string literal,
// prints the duplicate, and runs without errors
int main(int argc, char** argv) {
  char* sample = "Hello World!";

  char* dup;
  str_duplicate(sample, &dup);
  printf("%s", dup);
  free(dup);
  return EXIT_SUCCESS;
}
```

C++ Class

- Complete the IntList class which represents a linkedlist of integers (on Codio)

- In particular, complete the `remove_all(int)` function

C++ Class

- Complete the IntList class which represents a linkedlist of integers (on Codio)

- In particular, complete the `remove_all(int)` function

Answer uploaded to website under lecture slides as IntList.cc

Concurrency

- There are at least 4 bad practices/mistakes done with locks in the following code. Find them.
  - Assume `g_lock` and `k_lock` have been initialized and will be cleaned up.
  - Assume that these functions will be called by multi-threaded code.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

Concurrency

- k++ could have a data race on it
- k_lock is uncessarily used around a+=b
- g_lock is used when k_lock should be used
- cin >> c does not need to be locked, could cause significant delays.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

Scheduling

- Four processes are executing on one CPU following round robin scheduling:

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- You can assume:
  - All processes do not block for I/O or any resource.
  - Context switching and running the Scheduler are instantaneous.
  - If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

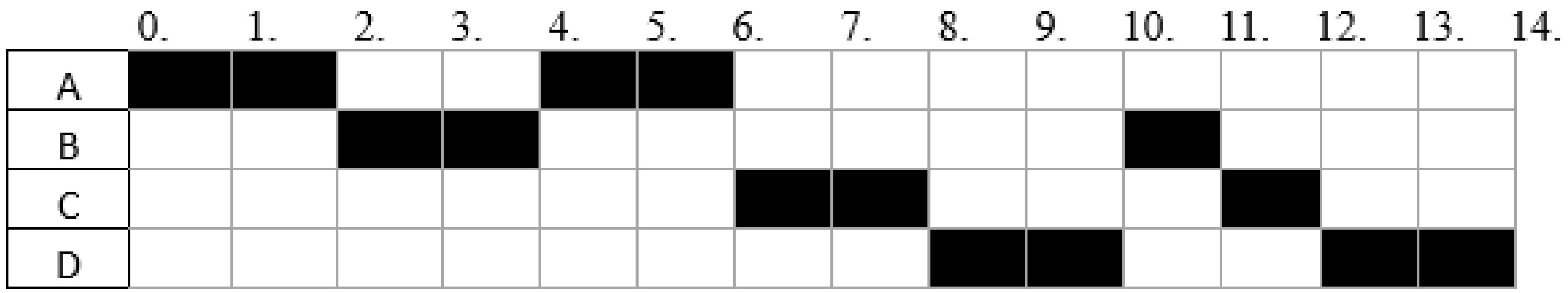
Scheduling

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

- What is the earliest time that process C could have arrived?
- Which processes are in the ready queue at time 9?
- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?

Scheduling



	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

- What is the earliest time that process C could have arrived?
  - If C arrived at time 0, 1, or 2, it would have run at time 4
  - C could have shown up at time 3 and come after A in the queue
  - C showed up at time 3 at earliest

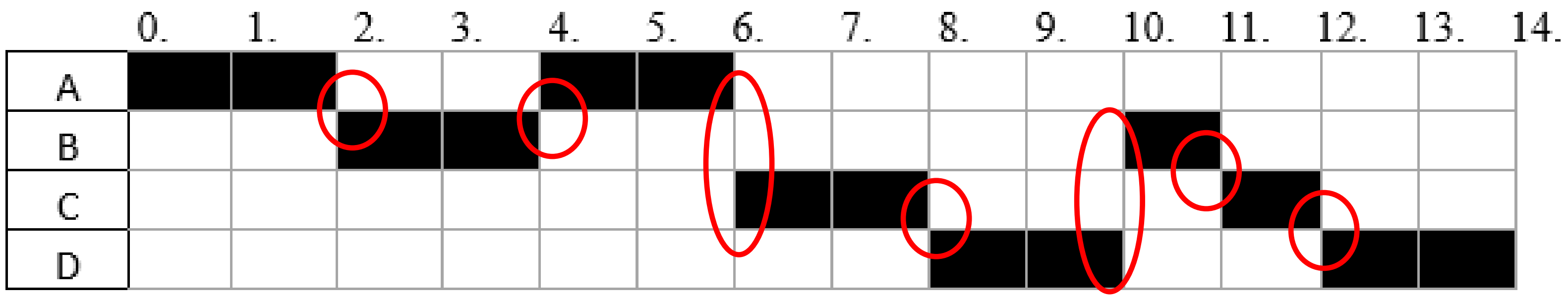
Scheduling

	0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
A	■	■			■	■									
B			■	■							■				
C							■	■				■			
D									■	■			■	■	

- All processes do not block for I/O or any resource.
- Context switching and running the Scheduler are instantaneous.
- If a process arrives at the same time as the running process' time slice finishes, the one that just arrived goes into the ready queue before the one that just finished its time slice.

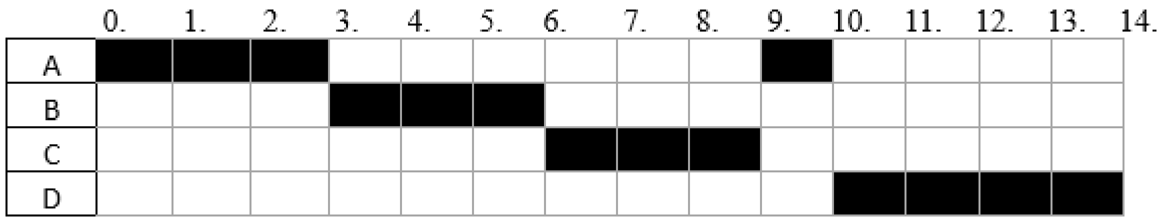
- Which processes are in the ready queue at time 9?
  - D is running, so it is not in the queue
  - A has finished
  - B and C still have to finish, so they are in the queue.

Scheduling

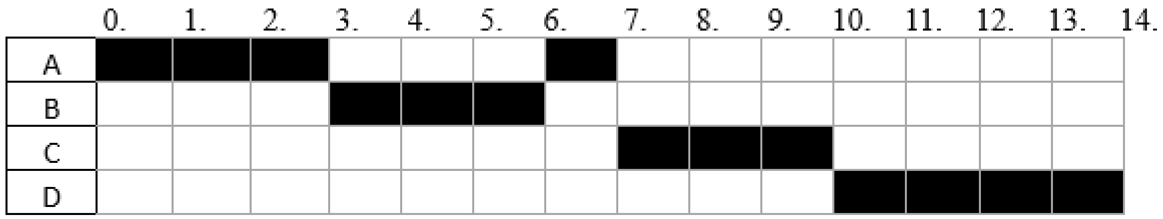


- If this algorithm used a quantum of 3 instead of 2, how many fewer context switches would there be?
  - Currently there are 7 context switches
  - If quantum was 3:

Depends on if C shows up at time 3 or 4



  - Or:



Either way, only 4 context switches, so 3 less than quantum = 2

Virtual Memory

- Consider a system with the following configuration:
  - 32-bit address space
  - 2GB physical memory size
  - 16-bit addressability (2 bytes per address)
  - 32KB page size

- How many entries are there in each process's page table? Express your answer as a power of 2.

- How many frames are there of physical memory?

Virtual Memory

- Consider a system with the following configuration:
  - 32-bit address space
  - 2GB physical memory size
  - 16-bit addressability (2 bytes per address)
  - 32KB page size

- How many entries are there in each process's page table? Express your answer as a power of 2.
  - 32-bit address space -> 2^{32} addresses. 2 bytes per address -> 2^{33} bytes in an address space.
  - 32KB = $32 * 2^{10} = 2^{15}$
  - $2^{33}/2^{15} = 2^{18}$ pages which is the number of entries in a page table

Virtual Memory

- Consider a system with the following configuration:
  - 32-bit address space
  - 2GB physical memory size
  - 16-bit addressability (2 bytes per address)
  - 32KB page size

- How many frames are there of physical memory?
  - 2 GB = 2 * 2^{30} -> 2^{31} bytes of physical memory
  - Each frame is 32KB -> 2^{15} bytes
  - 2^{31} / 2^{15} = 2^{16} frames

Caching (LRU)

- Consider that we have physical memory that can hold 4 pages of memory and uses LRU for replacement. Come up with an example scenario where having this policy hurts performance.

Caching

- Consider that we have physical memory that can hold 4 pages of memory and uses LRU for replacement. Come up with an example scenario where having this policy hurts performance.

- Consider the case where we had 5 pages of memory A, B, C, D, and E. We access them in order on a loop, so after we access page D we access Page E. Then we access page A, then page B…
- This results in the page we want to access never being in the cache
- This is called **thrashing**