Midterm Review

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- Midterm is coming soon (2 days from now!)
  - Meyerson B1 7:00 pm to 9:00pm Thursday 10/19
  - If you can't make the time, please send me an email **ASAP**
- Midterm Policies posted on the course website. Please read through them.
  - You are allowed 1 page of notes 8.5 x 11 double sided notes
  - Clobber policy: can show growth by doing better on the second midterm
- Recitation After lecture will be midterm review
- Lecture today will be scheduling, and then exam review
  - Thurs will continue the exam review we do today.

Midterm 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 provide parts of the man pages for various system calls)

- 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

Midterm 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)

Midterm 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

Midterm 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**

Topics

- Processes
- Page Tables
- Page Replacement Policy
- Memory Allocation
- Caches
- Threads

Processes

- We want to write a in C program that will compile and evaluate some other program. The program we are grading is similar to penn-shredder. For this program we write, lets assume we are running penn-shredder once and evaluating it. We need to be able to:
  - Specify the input and get output of the shredder
  - Set a time limit so that penn-shredder doesn't go infinite
  - Setup penn-shredder to receive signals from the keyboard (e.g. CTRL + C and CTRL + Z)

- Roughly how many times do we need to call each of these system calls? Briefly explain any system call you specify non-zero for

Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

System Call	Number	Justification
fork()		
execvp()		
pipe()		
waitpid()		
kill()		
signal()		
tcsetpgrp()		

Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

System Call	Number	Justification
fork()	2	Need to fork compiler and penn-shredder
execvp()	2	To exec compiler and penn-shredder
pipe()		
waitpid()		
kill()		
signal()		
tcsetpgrp()		

Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

System Call	Number	Justification
fork()	2	Need to fork compiler and penn-shredder
execvp()	2	To exec compiler and penn-shredder
pipe()	2	To send input and get output from penn-shredder
waitpid()		
kill()		
signal()		
tcsetpgrp()		

Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

System Call	Number	Justification
fork()	2	Need to fork compiler and penn-shredder
execvp()	2	To exec compiler and penn-shredder
pipe()	2	To send input and get output from penn-shredder
waitpid()	2	To wait for compiler and penn-shredder to terminate
kill()		
signal()		
tcsetpgrp()		

Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

Some of these can have varying answers. If this was an exam, as long as they are reasonable and justified well, we will accept it

System Call	Number	Justification
fork()	2	Need to fork compiler and penn-shredder
execvp()	2	To exec compiler and penn-shredder
pipe()	2	To send input and get output from penn-shredder
waitpid()	2	To wait for compiler and penn-shredder to terminate
kill()	1	Debatable, can be justified if you used it. I use it to kill the child after timeout has occurred. Better than just using alarm in child since we can handle the timeout more elegantly and print out an error
signal()		
tcsetpgrp()		

Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

Some of these can have varying answers. If this was an exam, as long as they are reasonable and justified well, we will accept it

System Call	Number	Justification
fork()	2	Need to fork compiler and penn-shredder
execvp()	2	To exec compiler and penn-shredder
pipe()	2	To send input and get output from penn-shredder
waitpid()	2	To wait for compiler and penn-shredder to terminate
kill()	1	… (trimmed for space see previous slides)
signal()	1	Debatable again. Used to register SIGALRM for timeout. Could be avoided if we register alarm in child
tcsetpgrp()		

Processes Cont.

- Roughly how many times do we need to call each of these system calls? Briefly explain your answer for every system call.

Some of these can have varying answers. If this was an exam, as long as they are reasonable and justified well, we will accept it

System Call	Number	Justification
fork()	2	Need to fork compiler and penn-shredder
execvp()	2	To exec compiler and penn-shredder
pipe()	2	To send input and get output from penn-shredder
waitpid()	2	To wait for compiler and penn-shredder to terminate
kill()	1	… (trimmed for space see previous slides)
signal()	1	… (trimmed for space see previous slides)
tcsetpgrp()	1	Debatable again. used so penn-shredder has control of terminal and so it will get the keyboard signals and our program won't. Could instead register the signals in our program with signal and use kill in handler to send to child.

Page Tables Q1

- One oddity about page tables is that the page table itself exists in memory. However, the memory that is used to store some page tables are usually “pinned” into memory, meaning that those page tables cannot be evicted/removed from physical memory.

- Why is it important that some of the memory representing these page tables remain “pinned”? Please explain your answer.

Page Tables Q1

- One oddity about page tables is that the page table itself exists in memory. However, the memory that is used to store some page tables are usually "pinned" into memory, meaning that those page tables cannot be evicted/removed from physical memory.

- Why is it important that some of the memory representing these page tables remain "pinned"? Please explain your answer.

Page tables exist in virtual memory, meaning we may need to do a lookup of the address of nodes in the page table. If we don't have some addresses pinned or specially handled, we could not do translations since we wouldn't know what physical memory contains the page table entries we need

Page Tables Q2

- When we first brought up the idea of page tables, we imagined the page table as one giant array containing one page table entry for each page. We investigated other page table implementations (inverted and multi-level) since this "big array" model uses up A LOT of space for entries that may never be used.

- Let's say we had a virtual page number that we wanted to translate to a physical page number. How would the lookup speed of our original "big array" page table model compare to the more space efficient page tables implementations?

Page Tables Q2

- When we first brought up the idea of page tables, we imagined the page table as one giant array containing one page table entry for each page. We investigated other page table implementations (inverted and multi-level) since this "big array" model uses up A LOT of space for entries that may never be used.

- Let's say we had a virtual page number that we wanted to translate to a physical page number. How would the lookup speed of our original "big array" page table model compare to the more space efficient page tables implementations?

It would be constant time lookup and only one memory access. We can index into the page table using the virtual page number we want to ranslate

Page Tables Q3

- One thing that is different about inverted page tables is that the page table has one entry per physical page instead of per virtual page.

- Because of this, a page table can be shared across all processes instead of being per process. This is since all processes share physical memory.

- If a page table is shared across all processes, what issues could this cause? How does an inverted page table handle this issue?

Page Tables Q3

- One thing that is different about inverted page tables is that the page table has one entry per physical page instead of per virtual page.
- Because of this, a page table can be shared across all processes instead of being per process. This is since all processes share physical memory.
- If a page table is shared across all processes, what issues could this cause? How does an inverted page table handle this issue?

We need to make sure processes only use memory allocated to that process. Inverted page table also stores the process ID with each entry and uses it in the hash to make sure only processes with the specified ID accesses that entry

Page Replacement Policy

- Seungmin and Nate are debating the best page replacement policy. One of them says that LRU is strictly better (e.g. better in all cases) than FIFO page replacement and always leads to less page faults.

- Is this true or false? Please explain your answer. If it is not true, provide an example of page accesses that counters this claim.

Page Replacement Policy

- Seungmin and Nate are debating the best page replacement policy. One of them says that LRU is strictly better (e.g. better in all cases) than FIFO page replacement and always leads to less page faults.

- Is this true or false? Please explain your answer. If it is not true, provide an example of page accesses that counters this claim.

False: consider we have 4 physical pages and have the reference string:
0 1 2 3 0 4 1 2 3
In LRU we get 8 page faults
In FIFO we get 5 page faults

Memory Allocation Q1

- Slab allocator is really fast, but it would be inconvenient to replace malloc with a slab allocator. Why is that?

- How much internal and external fragmentation does a slab allocator have?

Memory Allocation Q1

- Slab allocator is really fast, but it would be inconvenient to replace malloc with a slab allocator. Why is that?

Slab allocator only handles allocations of a specific size. If we replaced malloc with it, we could not handle allocations of all sizes. Allocation requests that are too big would not work and allocations of a small size would have a lot of internal fragmentaiont

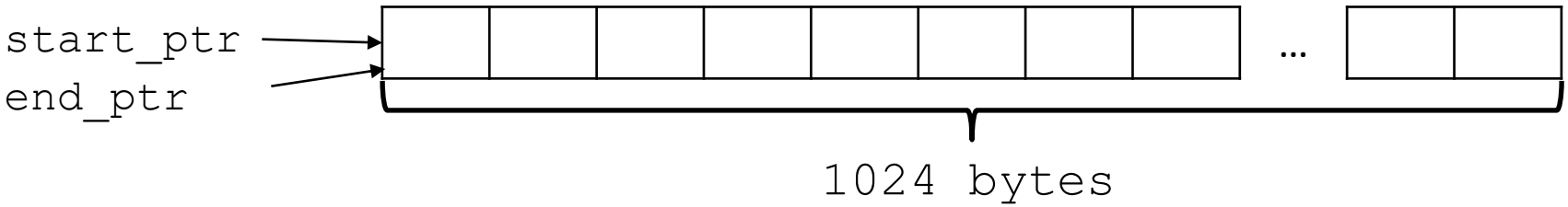
- How much internal and external fragmentation does a slab allocator have?

Minimal/none for both ☺ Since we know how big each allocation is, we can allocate the exact size requested (no internal) and chunk our memory so that there is minimal space between each allocated chunk

Memory Allocation Part 2

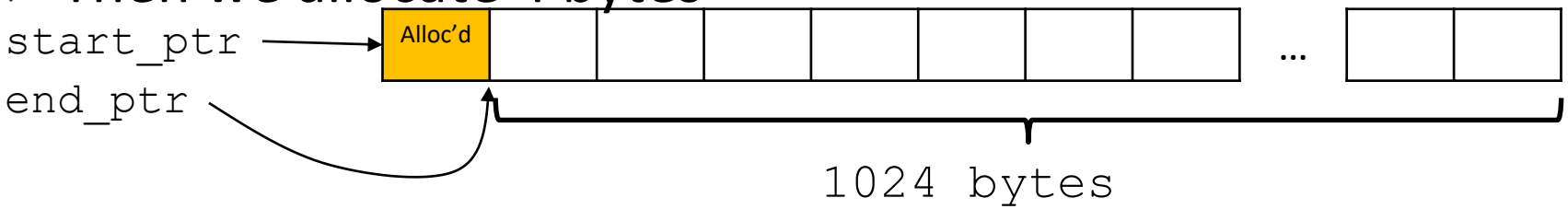
- In some instances, we want to allocate a lot of items and limit those allocations to one scope. We call our allocator a “temp_allocator” since it allocates things that are expected to be temporary to some scope.

- For example, Consider we start with:



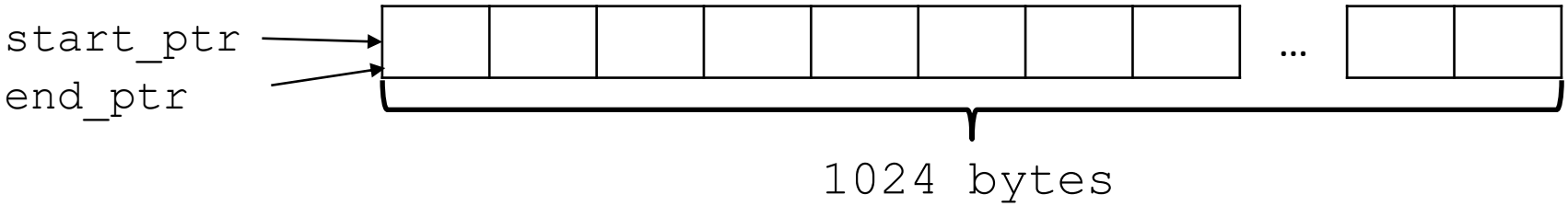
  - Note that there is no metadata, just these two pointers

- Then we allocate 4 bytes



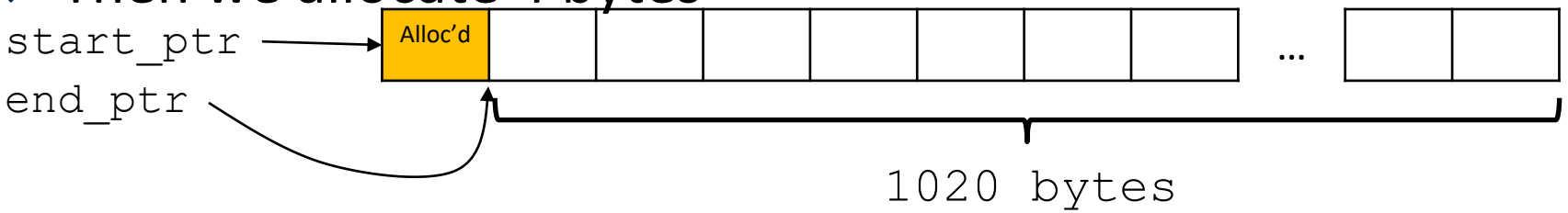
Memory Allocation Part 2

- For example, Consider we start with:

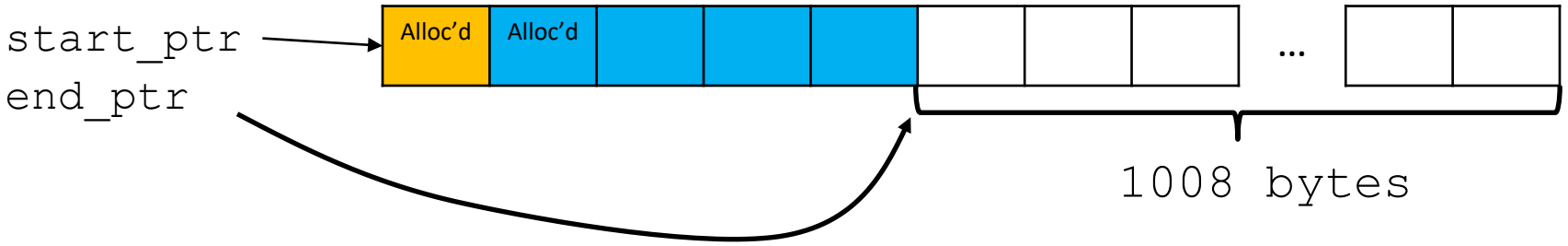


  - Note that there is no metadata, just these two pointers

- Then we allocate 4 bytes

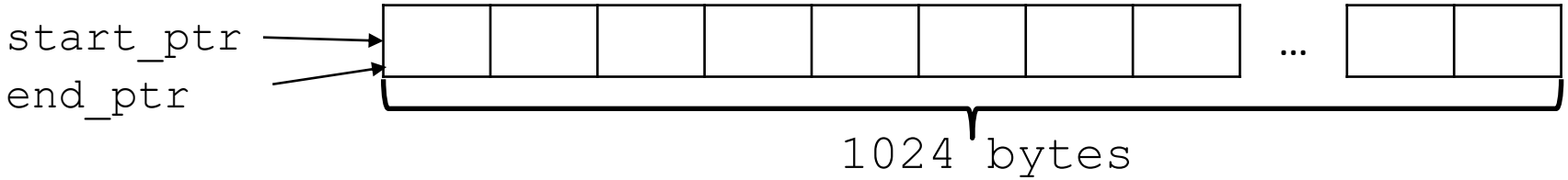


- Then we allocate 16 bytes



Memory Allocation Part 2

- Once we are done with our temporaries, we free the all allocations, and we can then use it again as if "fresh"



  - Looks the same as when we started!
- **That is the entire API**
- Example usage:

```
temp_allocator t_alloc = init_allocator();
for (many iterations) {
  int *ptr = allocate(t_alloc, 4 bytes);
  image *img = allocate(t_alloc, 1024 bytes);
  // a bunch of other allocations local
  // to this scope
  clear_allocs(t_alloc);
}
```

Memory Allocation Part 2

- How fast is our allocator at allocating things on average? At freeing things?
- What does the internal and external fragmentation look like with our allocator?
- Why can't we use this as a replacement for malloc maintaining lists of allocated & freed memory?

Memory Allocation Part 2

- How fast is our allocator at allocating things on average? At freeing things?

  Very Fast, constant time for each

- What does the internal and external fragmentation look like with our allocator?

  Minimal/none for both ☺ Since we know how big each allocation is, we can allocate the exact size requested (no internal) and chunk our memory so that there is minimal space between each allocated chunk

- Why can't we use this as a replacement for malloc maintaining lists of allocated & freed memory?

  Malloc manages things that are freed individually that may be allocated for varying lengths of time. This allocator assumes everything can be allocated together.

Caches Q1

- Let's say we are making a program that simulates various particles interacting with each other. To do this we have the following structs to represent a color and a point

```
struct color {
  int red, green, blue;
};
```

```
struct point {
  double x, y;
  struct color c;
};
```

- If we were to store 100 point structs in an array, and iterate over all of them, accessing them in order, roughly how many cache hits and cache misses would we have?
  - Assume:
    - a cache line is 64 bytes
    - the cache starts empty
    - `sizeof(point)` is 32 bytes, `sizeof(color)` is 16 bytes

Caches Q1

- Let's say we are making a program that simulates various particles interacting with each other. To do this we have the following structs to represent a color and a point

```
struct color {
  int red, green, blue;
};
```

```
struct point {
  double x, y;
  struct color c;
};
```

- If we were to store 100 point structs in an array, and iterate over all of them, accessing them in order, roughly how many cache hits and cache misses would we have?
  - Assume:
    - a cache line is 64 bytes
    - the cache starts empty
    - `sizeof(point)` is 32 bytes, `sizeof(color)` is 16 bytes

Roughly every other time we access a point struct, it will already be in the cache. The other 50% of the time, it needs to be fetched from memory

Caches Q2

- Consider the previous problem with point and color structs.

- In our simulator, it turns out a VERY common operation is to iterate over all points and do calculations with their X and Y values.

- How else can we store/represent the point objects to make this operation faster while still maintaining the same data? Roughly how many cache hits would we get from this updated code?

Caches Q2

- Consider the previous problem with point and color structs.
- In our simulator, it turns out a VERY common operation is to iterate over all points and do calculations with their X and Y values.
- How else can we store/represent the point objects to make this operation faster while still maintaining the same data? Roughly how many cache hits would we get from this updated code?

Change point to just be:

```
struct point {
  double x, y;
}
```

Then Store two arrays:

```
point arr1[100];
color arr2[100];
// point at index I
// has color arr2[i]
```

Each time we access a point, we can now load 4 points into the cache. We now get ~25 cache misses and 75 hits

Threads

- We have seen three concurrency models so far
  - Forking processes (fork)
    - Creates a new process, but each process will have 1 thread inside it
  - Kernel Level Threads (pthread_create)
    - User level library, but each thread we create is known by the kernel
    - 1:1 threading model
  - User Level Threads (ucontext_t)
    - We create threads at user level, leave them unknown to the OS and schedule them ourselves
    - N:1 threading model

Threads

- For each of the three concurrency models, state whether it is possible to do each of the following.
- In real exam, I would ask you to briefly explain why

	Processes	pthread	ucontext_t
Can share files and concurrently access those files.			
Can communicate through pipes			
Run in parallel with one another (assuming multiple CPUs/Cores)			
Modify and read the same data structure that is stored in the heap			
Switch to another concurrent task when one makes a blocking system call.			

Threads

- For each of the three concurrency models, state whether it is possible to do each of the following.
- In real exam, I would ask you to briefly explain why

	Processes	pthread	ucontext_t
Can share files and concurrently access those files.	Yes	Yes	Yes
Can communicate through pipes			
Run in parallel with one another (assuming multiple CPUs/Cores)			
Modify and read the same data structure that is stored in the heap			
Switch to another concurrent task when one makes a blocking system call.			

Threads

- For each of the three concurrency models, state whether it is possible to do each of the following.
- In real exam, I would ask you to briefly explain why

	Processes	pthread	ucontext_t
Can share files and concurrently access those files.	Yes	Yes	Yes
Can communicate through pipes (can't redirect w/o affecting other threads though)	Yes	Yes	Yes
Run in parallel with one another (assuming multiple CPUs/Cores)			
Modify and read the same data structure that is stored in the heap			
Switch to another concurrent task when one makes a blocking system call.			

Threads

- For each of the three concurrency models, state whether it is possible to do each of the following.
- In real exam, I would ask you to briefly explain why

	Processes	pthread	ucontext_t
Can share files and concurrently access those files.	Yes	Yes	Yes
Can communicate through pipes	Yes	Yes	Yes
Run in parallel with one another (assuming multiple CPUs/Cores)	Yes	Yes	No
Modify and read the same data structure that is stored in the heap			
Switch to another concurrent task when one makes a blocking system call.			

Threads

- For each of the three concurrency models, state whether it is possible to do each of the following.
- In real exam, I would ask you to briefly explain why

	Processes	pthread	ucontext_t
Can share files and concurrently access those files.	Yes	Yes	Yes
Can communicate through pipes	Yes	Yes	Yes
Run in parallel with one another (assuming multiple CPUs/Cores)	Yes	Yes	No
Modify and read the same data structure that is stored in the heap	No	Yes	Yes
Switch to another concurrent task when one makes a blocking system call.			

Threads

- For each of the three concurrency models, state whether it is possible to do each of the following.
- In real exam, I would ask you to briefly explain why

	Processes	pthread	ucontext_t
Can share files and concurrently access those files.	Yes	Yes	Yes
Can communicate through pipes	Yes	Yes	Yes
Run in parallel with one another (assuming multiple CPUs/Cores)	Yes	Yes	No
Modify and read the same data structure that is stored in the heap	No	Yes	Yes
Switch to another concurrent task when one makes a blocking system call.	Yes	Yes	No