NWEN 301 Project 2
Kandice McLean 300138073

Data Structures

thread.h

struct thread

int base_priority
This is used to store the threads base priority. This is so that if the threads priority is changed through priority donation, there is a reference to the base value so that it can easily be changed back. This base priority can only be changed through the thread_set_priority function, which is not called anywhere in the four files that were changed in this project.

Struct lock *wait_on_lock
This stores the lock that a thread is waiting on. This variable is very important in priority donation, as it is used to access the thread it is holding to donate to it. In pintos, there is a 4Kb page limit for a thread, and if you pass another thread as a struct into a function from another file (e.g from synch.c) in order to change its priority, the extra space taken up by the passed in thread causes the limit to be exceeded and a kernel panic to occur. This variable allows us to get around this, as the thread only a holds a lock, which is a relatively small struct and doesn't cause the thread to exceed its page limit, but from this lock the thread that needs the donation can be accessed by using struct thread *t = thread_current()->waiting_on_lock->holder.

Struct list donations
This struct is a list of all the other threads that are being blocked by this thread due to it holding a lock that they require. The list is kept sorted in order of priority.

Struct list_elem donation_elem
This struct is a wrapper for the threads stored in the donations list so that they can be manipulated through the kernel library list.

Algorithms

thread.c

tid_t thread_create (const char *name, int priority, thread_func *function, void *aux)
```
if(thread_current()->priority < priority){
   thread_yield();
}
```
If the priority of the new thread is greater that the priority of the currently running thread the current thread needs to yield to the new one, so added this if statement to deal with that situation.

Void thread_unblock (struct thread *t)
```
list_insert_ordered(&ready_list, &t->elem,(list_less_func *) &cmp_priority, NULL);
```
Void thread_yield (void)
```
list_insert_ordered(&ready_list, &t->elem,(list_less_func *) &cmp_priority, NULL);
```
The original implementation of these functions was to simply push the unblocked thread to the back of the ready_list. However, the list needs to be kept in order of priority as the current implementation pulls the first thread of the queue to run under the assumption that it is sorted, so updated to use the list_insert_ordered function with a comparator which was created that compares priorities passed in.

```
Void thread_set_priority (int new_priority){
  int old_priority = thread_current()->priority;
  thread_current ()->base_priority = new_priority;
  refresh_priority();
  if (old_priority > thread_current()->priority){
     thread_yield();
  }
}
```
The original implementation of this function was to simply set the current threads priority to the new priority. This need to be updated in two ways. First, since priority donation required the base_priority to be kept as a separate variable to priority, this function had to be updates so that base_priority would change rather than the priority, as this function is

only called if the base priority needs to be changed. The priority then needs to be refreshed. Second, the if statement was added to deal with the situation where the current priority is more than the new priority, which could mean that it is no longer the highest priority thread, so must yield to be reinserted into the queue in its correct place.

```
void refresh_priority (void){
  struct thread *current = thread_current();
  current->priority = current->base_priority;
  if (list_empty(&current->donations)) {
     return;
  }
  struct thread *best_donated_priority = list_entry(list_front(&current->donations), struct thread, donation_elem);
  if (best_donated_priority->priority > current->priority) {
     current->priority = best_donated_priority->priority;
  }
}
```

This function is used to refresh a threads priority. It deals with two separate situations. The first is where the thread has released all of its locks, so needs its priority to be refreshed back to its old priority. The function will do this if the list of donations is empty, meaning no threads are waiting on it. The second situation is if the thread is still holding a lock. If this happens, it checks the best available priority in the donations list and if the donated priority is better it will receive the donation. The list of donations is ordered by priority so it simply needs to check the head of the queue.

```
Static void init_thread (struct thread *t, const char *name, int prioriy)
t->base_priority = priority;
t->wait_on_lock = NULL;
list_init(&t->donations);
```

These lines were added to initialise the new variables for priority donation in struct thread.

```
bool cmp_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED){
  struct thread *t1 = list_entry(a, struct thread, elem);
  struct thread *t2 = list_entry(b, struct thread, elem);
  return (t1->priority > t2->priority);
}
```

This is a comparator that returns true if thread a has a higher priority than thread b. It is used for keeping ready_list and donations sorted by priority in thread.c and semaphore wait-lists in synch.c. Because it is used for a list, the parameters passed in must be list_elem, then pulled out into the thread structure. Passing thread structs in as parameters would make this comparator incompatible with the list structure in kernel library.

```
void donate_priority (void){
  struct thread *current = thread_current();
  struct lock *wanted_lock = current->wait_on_lock;
  while (wanted_lock!=NULL){
     if (!wanted_lock->holder){
              return;
            }
     if (wanted_lock->holder->priority >= current->priority){
              return;
            }
     wanted_lock->holder->priority = current->priority;
     current = wanted_lock->holder;
     wanted_lock = current->wait_on_lock;
    }
}
```

Although a form of priority donation is dealt with in the refresh priority function, this function also needed to be created to deal with a situation that occurs when multiple threads are waiting on locks. What can happen is that thread A is waiting on a lock held by thread B, who has received a donation from thread A but still cannot proceed as they are now waiting on thread C's lock. The refresh function deals with thread A and B, but as C is neither the current thread (A) nor the thread holding the lock that A wishes to receive, refresh_priority will not update C's priority, so all three threads remain blocked. The function starts of by dealing with the situation of thread A and B. Once it has fixed their priorities, it will treat thread B as the current thread and fix thread C. Then if there is a thread D that C is waiting on it will treat C as current, and so on, until one of three things happen that will signal all threads are ready to go again; it reaches a thread that is not waiting on a lock, the lock the current thread wants is free, or the thread occupying the wanted lock of the current thread has a higher priority than the current thread. Once one of these conditions have been met this indicates the threads are all unblocked and the system can move on.

void remove_with_lock(struct lock *lock){

```
 struct list_elem *e = list_begin(&thread_current()->donations);
 struct list_elem *next;
 while (e != list_end(&thread_current()->donations))
  {
    struct thread *t = list_entry(e, struct thread, donation_elem);
    next = list_next(e);
    if (t->wait_on_lock == lock){
            list_remove(e);
          }
    e = next;
  }
```

}

Although a lock can only hold one thread at a time, a thread may hold multiple locks. Consider the situation where a thread is both reading from the cpu and writing at the same time due to memory restrictions. In this case, it will be holding two locks, one for each process. This function is required as when releasing a lock, it is necessary to clear the threads waiting for the lock out of the donations list. However, if the thread has released the reading lock, but is still holding the writing lock, the donators waiting for the writing lock need to remain in the donations list, otherwise the donations for the writing lock would not be implemented, and the refresh_priority function would see an empty list and change the threads priority back to its base_priority, further affecting donations. To prevent this, this function checks each donator to make sure the lock they were waiting for was the lock that was released before removing the donator from the list.

Synch.c

void sema_down (struct semaphore *sema)

donate_priority();

This was added to implement semaphore priority. It uses donate_priority rather than refresh_priority because there may be more than one thread blocking the current one from running. Originally the insertion of the thread into the queue was an ordered one, but the tests showed it wasn't remaining sorted after implementing priority donation. In order to reduce complexity this was changed back to the original push_back implementation and sorting is handled in sema_up before unblocking instead.

Void sema_up (struct semaphore *sema)

```
 if (!list_empty (&sema->waiters)) {
    list_sort(&sema->waiters, (list_less_func *) &cmp_priority, NULL);
    cur = (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
    thread_unblock(cur);
  }
  sema->value++;
  if (thread_current()->priority < cur->priority) {
    thread_yield();
  }
```

Two things have been updated here. First, the list is now sorted by priority before unblocking the head of the queue. This ensures the highest priority thread is at the head of the queue when it is popped off and unblocked. This top thread is also assigned to a variable so it can be used in the second part of the function. The second part checks if the currently running thread is of a higher priority than the one being unblocked, if it is not, then the current thread yields.

Void lock_acquire (struct lock *lock)

```
 enum intr_level old_level = intr_disable();
  if (lock->holder){
    thread_current()->wait_on_lock = lock;
    list_insert_ordered(&lock->holder->donations, &thread_current()->donation_elem, (list_less_func *)
&cmp_priority, NULL);
  }
  sema_down (&lock->semaphore);
  thread_current()->wait_on_lock = NULL;
  lock->holder = thread_current ();
  intr_set_level(old_level);
```

If the lock has a holder, the current thread will assign the lock to its wait_on_lock variable, then will be inserted into the holders donations list in order of priority. sema_down is then called, which contains the call to donate_priority. After sema_down has finished, the thread can acquire the lock, so it sets its wait_on_lock back to NULL and becomes the holder of the lock.

Bool lock_try_acquire (struct lock *lock)
thread_current()->wait_on_lock = NULL;
If the thread is successful at acquiring the lock, it sets its wait_on_lock to NULL.

Void lock_release (struct lock *lock)
remove_with_lock(lock);
refresh_priority();
This particular implementation of pintos has a list of donator threads contained in the thread structure used for priority donation. The donators associated with the lock need to be removed once the lock is released, as they are no longer required. After these donators are removed, priority is refreshed, which will return its priority to base if no other locks are currently held.

Void cond_signal (struct condition *cond, struct lock *lock UNUSED)
*list_sort(&cond->waiters, (list_less_func *) &cmp_sem_priority, NULL);*
As the waiters for a conditional variable are simply pushed to the back of the list when added in the cond_wait function, the list needs to be sorted to ensure the thread at the front is of the highest priority, as this is the one which is woken. Sorting is done instead of inserting in order, as inserting in order seemed to only pass the tests if the list is sorted in the signal. To reduce complexity, the original implementation of pushing to the back of the list was reintroduced and keeping the list sorted was left to cond_signal.

bool cmp_sem_priority (const struct list_elem *a, const struct list_elem *b, void *aux UNUSED){
```
  struct semaphore_elem *s1 = list_entry(a, struct semaphore_elem, elem);
  struct semaphore_elem *s2 = list_entry(b, struct semaphore_elem, elem);
  if ( list_empty(&s2->semaphore.waiters) ){
     return true;
  }
  if ( list_empty(&s1->semaphore.waiters) ){
     return false;
  }
  struct thread *t1 = list_entry(list_front(&s1->semaphore.waiters), struct thread, elem);
  struct thread *t2 = list_entry(list_front(&s2->semaphore.waiters), struct thread, elem);
  return (t1->priority > t2->priority);
```
}
This comparator is similar to the one in thread.c, except it compares two semaphores instead of two threads. It will return true if either the second semaphores list of waiting threads is empty or if the top priority thread in semaphore 1s wait list is higher priority than the highest priority thread in semaphore 2. This comparator only works on the assumption that the list of waiting threads in the semaphores is correctly ordered by priority.