// name:Benjamin Weiss bsw46, Xingyu Yao xy139, Ryan Shah rds216
// username of iLab:cd.cs.rutgers.edu
// iLab Server:cd.cs.rutgers.edu

Structure definition:

threadControlBlock(tcb): contains an integer variable id as the thread id, an ucontext_t variable uc as the thread context, and a void pointer return_value as the return value of thread.

my_pthread_mutex_t: contains an integer variable mid as the mutex id, an integer variable lock as the lock mechanism, and a integer variable T_hold as the holder of mutex.

A multilevel queue called ready queue is used to store threads that are ready to run and manage priority. Ready queue is a array of five queues each represent a priority.
A linked list of queue called wait queue is used to manage mutexes while each queue corresponds to a mutex and contains threads are are waiting to get the key for the specific mutex.

node: the node for queue. Contains a node pointer to the next node, an tcb pointer representing thread and an integer variable level to store priority.

queue: queue for ready and wait queue. Contains two node pointers front and back as the head and rear node of queue, integer variables size as size of queue, level to store priority, multiplier as the multuplier of priority and thread_done as threads finish execution.

ready_queue: an array of queues of five as the ready queue.

waitQueues: the linked list of queues as the wait queue. Contains an integer variable as id of mutex, node pointer as the head of queue and waitQueue pointer to the next wait queue in linked list.

kilist: a linked list of node that store the threads that finish execution

Global Variable:

- ucontext_t
  - thr: context used in thread creation as new thread
  - main_context: context of main
  - op_context: the returen context of new thread in thread creation
- killist: the kilist
- tcb * current: the current thread executing
- ready_queue readyQ : the ready queue
- int
  - level: the current priority in ready queue
  - Sys: indicator of atomic build-in for accessing system/user mode

waitQueue * waitQ: the waitqueue
waitQueue * current_wait_queue: the current executing wait queue(mutex)

Functions

multilevelQueue: initialize the global ready queue pointer as a multilevel queue as ready queue.
start_itimer: initialize the itimer and send signal SIGPROF every 25 milisecond.
stop_itimer: stop the itimer.
Signal handler: compare_and_swap into system mode and stop itimer. schedule the threads' priority in the ready-queue by dequeue the target thread from its ready queue and enqueue it into the next priority queue before yield. Swap context. Compare_and_swap into user mode and start itimer.
my_pthread_exit: compare_and_swap into system mode and stop itimer.
End the current thread, dequeue it from ready_queue the enqueue into the kilist
Do a seperate case for main context in that the whole program will end.
Checked logic it is for, when exiting a queue reaches priority limit due to the thread being elminiated counting as a thread done. This leads to switching to the next.
Compare_and_swap into user mode and start itimer.
my_pthread_create: compare_and_swap into system mode and stop itimer. If the ready_queue is empty, initialize the multilevel queue and enqueue the main context as current thread. Build the node for current and new thread. store them into ready_queue after context switch then yield. Compare_and_swap into user mode and start itimer.
enqueue: add a node to the rear of queue
dequeue: pop a node to the rear of queue
my_pthread_yield: compare_and_swap into system mode and stop itimer.
dequeue current thread from the ready_queue and switch current thread the new thread's position. Context switch and change their priority indicator.
Compare_and_swap into user mode and start itimer.
my_pthread_join: yield children threads and dequeue them from ready_queue and enqueue into kilist. Store the pointer of return value from yield as the return value of join. Join function takes o(n) time for each run of it in the worst case.
my_pthread_mutex_init:
Compare_and_swap into system mode and stop itimer.initialize the wait queue.
Compare_and_swap into user mode and start itimer.
my_pthread_mutex_lock:
Compare_and_swap into system mode and stop itimer.Putting current thread into wait queue. Storing into current wait Q.traverse the current wait Q to enqueue new node into the rear of wait Q and context switch. enqueue current thread into waitqueue after dequeue it from the ready queue. Compare_and_swap into user mode and start itimer.
my_pthread_mutex_unlock:
Compare_and_swap into system mode and stop itimer. Reset lock back into 0 and dequeue a thread from the wait queue and enqueue back into its priority queue. Compare_and_swap into user mode and start itimer.

my_pthread_mutex_destroy:
Compare_and_swap into system mode and stop itimer. If the corresponding wait q is Null and mutex is unlock, free queue. Or EXIT_FAILURE. Compare_and_swap into user mode and start itimer.

my_pthread_mutex_search:
Compare_and_swap into system mode and stop itimer. Seach for the current mutex’s wait queue in the linkedlist of waitqueue, return it as he global variable current_wait_queue. Compare_and_swap into user mode and start itimer.

test case used:

```
void* myfunc(void* a){
//printf("%d:%d\n", current->id,readyQ->queues[0]->size);
printf("billy\n");
my_pthread_mutex_lock(key);
printf("yes!!!\n" );
while(1){}
my_pthread_mutex_unlock(key);

return;
}
void* myfunc2(void* b){
//printf("%d:%d\n", current->id,readyQ->queues[0]->size);
printf("billy\n");
my_pthread_mutex_lock(key2);
printf("yes\n" );
if(b!=NULL){
int* a=(int*)b;
if(*a==4){
printf("NOOO");
}
}
my_pthread_mutex_unlock(key2);
my_pthread_exit(b);
return;
}



int main(){
  // thread_init();
   //printf("%d\n",current->id );
//Createthread
   key=malloc(sizeof(my_pthread_mutex_t));
```

```
    pthread_mutexattr_t *mutexattr;
    my_pthread_mutex_init(key,mutexattr);
    key2=malloc(sizeof(my_pthread_mutex_t));
    my_pthread_mutex_init(key2,mutexattr);
 //void*=(void)42;
    my_pthread_t * thread=malloc(sizeof(my_pthread_t));
    printf("first\n");
    my_pthread_create(thread,NULL, (void *)(*myfunc),NULL);
    my_pthread_create(thread,NULL, (void *)(*myfunc),NULL);
    my_pthread_create(thread,NULL, (void *)(*myfunc),NULL);
    my_pthread_t * thread2=malloc(sizeof(my_pthread_t));
    printf("second\n");
int* a=malloc(sizeof(int));
*a=4;
void* jay=a;
int* b=malloc(sizeof(int));
*b=5;
void* bob=b;
    my_pthread_create(thread2,NULL, (void *)(*myfunc2),jay);
    my_pthread_create(thread2,NULL, (void *)(*myfunc2),bob);
    my_pthread_create(thread2,NULL, (void *)(*myfunc2),NULL);
    my_pthread_yield();
    my_pthread_yield();
    my_pthread_yield();
    my_pthread_yield();
    my_pthread_yield();
    my_pthread_mutex_destroy(key2);
void** james=(void**)malloc(sizeof(void*));

            my_pthread_join(5,james);
int** lap =(int**)james;
//char* s=atoi((**lap));
        printf("%d\n",(**lap));
            my_pthread_join(4,james);
            my_pthread_join(6,NULL);
int** ss =(int**)james;
    printf("%d,%d\n",waitQ->id,(**ss));
    return 0;

}
```

result: printed 5; 1,4