Design Of Project 2

15-412:Implementing A Thread Library
And The 412maze Game

Alok Menghrajani (amenghra@andrew.cmu.edu)
&
Vikram Manjunath (vikramm@cs.cmu.edu)

February 17, 2003

1 Deadlock Detection:

Following is a detailed description of our design for deadlock detection:
In a single `maze_request_move()`, if monsters A, B and C want to move to cells x, y and z, then we add all of A, B and C to the "wait" list of x, y and z. As long as a certain group of monsters is waiting (has not yet moved in or been told that it can't by `maze_request_move()`) on a certain cell, that group of monsters is kept in the "wait" list of that cell. Whenever a request is made, we check if adding the monsters to the lists of the requested cells is going to lead to a circular wait. We do this by doing a DFS (on the graph formed by viewing the monsters as nodes and the presence of a monster one another monster's current cell's list as an edge). We do this with a short recursive routine. Since it is a very tight and small routine, we believe that an optimizing compiler should be able to emit code that can recurse without too much overhead.

2 asm_spinlock.s

This provides simple aquire and release functions for spin locks.

`void get_spinlock(int *lock)`

This function blocks until the lock at address lock is obtained. It tries to do an exchange, insterting a 0 and waiting to receive a 1. It spins by yielding after a failure and then trying again when it is scheduled next.

`extern void put_spinlock(int *lock)` This function just exchanges (xchg in x86) a 1 into the lock.

`extern int asm_XCHG (int * lock, int value)` This simply gives the functionality of the xchg x86 instruction. It tries to exchange value into the address lock.

3 asm_sysminclone.s

`int asm_sysminclone(void *stack)` This is a wrapper for the minclone system call. It takes care of setting the stack pointer of the child process to the place pointed to by stack. It assumes that the default return address is set (More details in section thr.c). Also, it does not actually do a `call` in the child process. In stead, it assumes that the stack that is passed in is set up such that the top word is the argument for the child process, the second word is the default return address and the

third word is the the address which is just above the new `%esp` and is the address of the function to be called. So, this function simply does a `ret` and this goes to the appropriate function.

It also returns the child's id or -1 to the parent process, depending on whether or not it is successful. -1 is returned in the case of failure.

4 condition.c

This provides a condition variable package are specified in the assignment.

We implemented condition variables using a spinlock and a queue. The spinlock is to protect the queue datastructure.

`int cond_init(cond_t * cv)` Intializes a condition variable. This needs to be called before using the condition variabls since the internal datastructures need to be initialized.

`int cond_destroy(cond_t * mp)` This frees the memory that is allocatd during `cond_init`.

`int cond_wait(cond_t * cv,mutex_t * mp)` When a thread calls this, it is placed on cv's queue and the mutex mp is released just before the thread is put to sleep.

`int cond_wait_spin(cond_t * cv, int * spinlock)` This is identical to `cond_wait` except that it takes a pointer to a spinlock (int) instead of a mutex.

`int cond_signal(cond_t * cv)` Signals to the first process on the queue that the predicate has been satisfied and frees the memory that eas used for that node of the queue.

`int cond_broadcast(cond_t * cv)` Signals to all processes on the queue and destroys the queue.

5 maze.c

`int maze_init()` initializes all the datastructures that the game will need. This includes an array of cell_ts of size BOARD_WIDTH X BOARD_HEIGHT. Each cell_t has various feilds as follows:

```
typedef struct hippo_plays_pacman
  tile_t tile;
  monster_t * monster; // This points to
                       //the mosnter standing on that tile
  int lock;                // we have locks for each cell
  monster_t** reserve; // This is to make
                       // reservations through make_request_move
  int reserve_n;       // This is the  number of threads that have
                       // reserved that tile
  waitlist* deps;      // This is a queue of monsters waiting on the cell
  cond_t wait;         // The condition variable that monsters can wait on
 cell_t
```

`int maze_cleanup()` The datastructures that are intialized in init are cleaned up here.

`tile_t maze_get_board_contents(int x, int y)` As specified in the api. This returns the contents of the tile, be it an actual tile content or a monster standing there.

`int maze_set_board_contents(int x, int y, tile_t tile)` This changes the contents of the board. But it does not change a monster who is standing there.

`monid_t maze_new_monster(tile_t t, int x, int y)` Creates a monster on the cell (x,y) of the board, with the color t. Makes sure that no rules of the game such as not being on a wall are violated.

`int maze_destroy_monster(monid_t mon)` Removes the monster with monid mon from the master list of monsters. Also, removes the want edges and reservations made by this monster.

`int maze_move_monster(monid_t mon, direction_t dir)` Moves the monster to the appropriate cell if permitted. This checks that if there is a reservation on the destination cell, unless mon is the owner of that reservation, he is not allowed to make that move. In the case of a move actually taking place, the monsters who are waiting on the cell are woken up by a `cond_broadcast()` on the conditional variable of that cell.

`int maze_request_move(move_combo_t *ml, int n)` We check that for various illegal situations. If we don't return FAILURE, , we make insert a "want edge" by putting the monsters on the "wait" lists of the destination cells. Then, we check if this leads to a deadlock. If so, we remove the want edge and return ERR_LOCK. And then, we make a reservation for the cells, if they are not already reserved by other groups. We then check if the cells are free to move to and if they are, we return SUCCESS. Otherwise, we keep waiting on those cells to become free, relenquishing the reservations we made everytime we go to sleep and picking them up again when we wake up. * READ NOTE AT END ABOUT WHY OUR DESIGN IS LIKE THIS.

`void maze_set_game_status(int s)` This changes the variable `gane_status` under thread safe conditions.

`int maze_get_game_status(void)` returns the value of `game_status`.

`void maze_abort(void)` wakes up all the sleeping threads who called `maze_wait`.

`void maze_wait(void)` goes to sleep on a queue.

`int maze_consume(monid_t mon)` sets the cell on which mon is standing to EMPTY.

Note: that all of our global data structures are protected by spinlocks.

6 mutex.c

We have implemented our mutexes in a manner similar to pthreads mutexes rather than what the assignment asked for in specific. We do not use this anywhere in out code. We have used "spinlocks" instead, and these behave much like what is asked for in the assignment specification (in the form of mutexes). This mutex implementation has bounded waiting and uses a queue to decide which of the waiting threads should get the lock next.

7 thr.c

The main thread library.

```
typedef struct _thread
  struct _thread *next;        // Linked list of threads. The first one is
                                // referenced by threads_list.
  int tid;                      // Thread ID, same as process ID.
  void *stack;                  // Pointer to the thread's stack.
                                // This is actually a pointer to the bottom
                                // of the stack, since it's the
                                // pointer returned by malloc before we
                                // add the size of the stack to it (because
                                // stacks grow downwards)
  int joinerlock;               // A spinlock to avoid two people from
                                // joining on a given thread at the same time.
  queue_node * joiners;        // List of threads to wake up on exit
  void* exit_status;           // This is where the exit status is saved
                                // when the thread exits.
  int state;                    // The state of the thread, either running
                                // or zombie (exited, ready to be reaped)
  int runlock;                  // lock to avoid going to sleep right after a
                                // call to wake up.
  int exitlock;                 // exitlock is a spinlock to avoid a thread
                                // from being reaped before it has done with
                                // exiting.
 thread_t;
```

universal_joiners is a list of threads that have to be woken up when a any thread exits.

`int thr_init(unsigned int size)`

This routine should be called before creating any threads. mm_init should have been called before. Returns -1 is called more than once. Size is the size of the stacks for the threads. Usually 1KB or 2KB. thr_init "converts" the calling process into a thread (it creates a thread structure for it), so that it can be joined upon.

`int thr_create(void *(*func)(void *), void *args)`

Creates a new thread and runs func with argument args. It returns the tid of the created thread. Allocates and sets up a new stack for the thread. Right before callig sys_minclone, the stack looks like this:

args
@asm_default_exit
@func

This way we run the func by simply doing the ret instruction (line 35 of asm_sysminclone.s) The threads_list is locked before calling sys_minclone in order to insure that the parent will finish filling the thread_t structure before the new thread accesses it.

There is no private addressing space (0 bytes are reserved at the top of the stack).

`int thr_join(int tid, int *departed, void **status)` suspend execution until thread tid exits. If tid is 0, you will be woken up by any exiting thread (universal_joiner). The beheaviour when you are woken up will depend if you are on the universal_joiners list (tid=0) or a specific thread's joiners list (tid!=0).

If you are on the universal list (tid=0), and you don't find any threads to reap, you go back to sleep. If you are on a specific thread's joiners list, and that thread has already been reaped by

someone else, then you return an error.

`void thr_exit(void *status)` Simply wake up all the threads in universal_joiners and the threads joiners list and "atomically" unlock the exit_lock and call sys_exit (these two operations need not be atomic, but they need to be executed without relying on the stack, as it may no longer exist. So we wrote them in assembly, which avoid procedure call, and thus avoids the need to use a stack !)

`thread_t* thr_new()` simpy allocate memory for one thread_t structure.

`thread_t* find_thread_t(int tid)` Each time we need to access a thread's own thread_t we call find_thread_t with sys_get_pid() as parameter. This implies going through the entire linked list to find ownself. It turns out that there is no other easy way out, since we could save some data on the thread's private address (at the top of the stack), but finding the top of the stack isn't always trivial.

`int thr_getid()` same as sys_get_pid()

Note: we have assumed that only thread_safe_malloc and thread_safe_free (which uses an internal spinlock) that we have provided in `safe_malloc.c` will be called by the user of the package to allocate memory and no direct calls will be made.

8 A note on our design of deadlock detection:

Given that we did not follow the text book algorithm for deadlock detection, we were forced to try a few iterations before we came up with what we have now. Our initial design was to simply not draw a distinction between wanting and reserving a cell. Our final design was compliant with the game and it turns out to be very similar to what is in the text book and what was presented in class. We realise that there is a corner case (that will not be tested by this 412maze game. This is that if we have two calls to `maze_request_move()` Such that both calls ask for the same cells but in different order, then we could potentially end up in an undetected deadlock. The work around is simply to synchronize calls to `maze_request_move()` or to canonicalize the order in which we get the locks for the cells.