Sample Mid-Term Exam 2

CS 5460/6460, Fall 2009

Mid-term date: November 19

Name: _____

Instructions: You have 70 minutes to complete this open-book, open-note, closed-computer exam. Please write all answers in the provided space, plus the back of the exam if necessary.

1) [50 pts] Users of the MT library would like a new operation:

```
int MT_wait_or(sema_t *sem, int tid);
```

This function is a combination of `MT_sem_wait()` and `MT_join()`, where the function blocks until either a semaphore is available or another thread has terminated. The result is `1` if the function returns because of a successful wait on the given semaphore, and it returns `2` if the function returns because the given thread terminated. If the semaphore is available and the thread is terminated, then `MT_wait_or()` picks one or the other arbitrarily.

The following scenarios indicate some of the ways that these operation should work and interact with other MT functions, where "..." means that a thread is performing some unspecified work that does not use `sem`, while a blank space indicates that the thread is blocked or terminated:

Only thread ends:

thread 1	thread 2	thread 3
`MT_sem_init(&sem, 0)`	...	...
...	`MT_wait_or(&sema, 3)`	...
...		`MT_exit()`
...	*result here must be 2*	
...	...	

Only semaphore available:

thread 1	thread 2	thread 3
`MT_sem_init(&sem, 0)`	...	...
...	`MT_wait_or(&sema, 3)`	...
`MT_sem_signal(&sem)`		...
...	*result here must be 1*	...

Both available:

thread 1	thread 2	thread 3
`MT_sem_init(&sem, 0)`	...	...
...	`MT_wait_or(&sema, 3)`	...
`MT_sem_signal(&sem)`		`MT_exit()`
...	*result here can be 1 or 2*	
`MT_sem_wait(&sema)`	...	
continues if thread 2 got 2;	...	
blocked if thread 2 got 1	...	

Describe **using C code** changes to the solution for HW4 to add the `MT_wait_or()` operation. Portions of the HW4 solution are attached to the end of this exam, including all parts that might need to change, and you can write on those pages as part of your answer. Your answer does not have to be error-free C code, but it should be close to working code.

2) [25 pts] Given the sequence of page accesses in the top row below, and given three available frames, show the page contained in each frame for after each page access according to the FIFO, 2nd chance, LRU, and optimal replacement algorithms. Also, show total the number of page faults when using each algorithm.

Accesses:	1	2	3	1	4	1	2	3	1	4	1	
FIFO												Faults:
2nd chance												Faults:
LRU												Faults:
optimal												Faults:

3) [25 pts] Some file systems support filling a file with 0s without taking up blocks on the disk, since a long sequences of 0s can be represented compactly and easily. When data is latter written to part of the file containing 0s, an actual block must be allocated. This feature can be supported in the file system of HW6 by using "-1" as a block number to indicate a read-only block that is filled with 0s. As is typical, this zero-filling rule applies only for 0s added to a file through a "truncate" operation or "lseek" operation is used to extend the size of the file. (That is, `MT_fs_write()` doesn't trigger the use of the "-1" page even if the bytes to be written are all 0s.) Also, only when a range of 0s spans an entire block is the "-1" block used.

For reference, the part of the HW6 that defines the file system format is included at the end of this exam.

- Does this change affect the maximum useful size of a disk that uses (only) this file system? Why or why not?

- Are there any new failure modes that can affect a program that calls `MT_fs` functions? Why or why not?

- Describe how each of the operations `MT_fs_open()`, `MT_fs_close()`, `MT_fs_unlink()`, `MT_fs_read()`, `MT_fs_write()`, `MT_fs_getpos()`, `MT_fs_truncate()`, and `MT_fs_lseek()` need to change to support "-1" as a block number.

```
struct sema {
  int cnt;
  struct tcb *wait;
};
...
enum thr_state { READY = 3333, RUNNING, WAITING, TERMINATED };
...
struct tcb {
  ...
  struct tcb *prev, *next;     // linked list pointers
  struct tcb *all_next;        // another linked list
  struct tcb *wait_q;          // queue of threads that have joined this thread
  int share;
  long long int vt;
  long long int start;
  long long int wake_time;
  ...
};
static struct tcb *ready_q;
static struct tcb *timer_q;
...
static int is_on_queue (struct tcb **q, struct tcb *t);
static void add_tail (struct tcb **q, struct tcb *t);
static struct tcb *dequeue_head (struct tcb **q);
static void remove_queue (struct tcb **q, struct tcb *t);
...
static void make_ready (struct tcb *t)
{
  add_tail (&ready_q, t);
  t->state = READY;
}
...
static void wake_up (struct tcb *t)
{
  t->vt = maxll (t->vt, get_min_vt ());
  make_ready (t);
}
...
static void block_me (void)
{
  int res_v;

  current->state = WAITING;
  update_current_vt ();

  save_context (&res_v);
  if (res_v == 0) {
    current = NULL;
    dispatch ();
  }
}
```

```
...
static void check_for_expired_timers (void)
{
  long long int now = get_real_time ();

 startover:

  check_invariants();

  if (timer_q) {
    struct tcb *t = timer_q;
    do {
      if (now >= t->wake_time) {
        remove_queue (&timer_q, t);
        wake_up (t);
        goto startover;
      }
      t = t->next;
    } while (t != timer_q);
  }
}
...
void MT_sem_wait (struct sema *sem)
{
  enter_MT_kernel ();

  while (sem->cnt <= 0) {
    add_tail (&sem->wait, current);
    block_me ();
  }

  sem->cnt--;

  exit_MT_kernel ();
}
...
void MT_sem_signal (struct sema *sem)
{
  struct tcb *p;

  enter_MT_kernel ();

  p = dequeue_head (&sem->wait);
  if (p)
    wake_up (p);
  sem->cnt++;

  exit_MT_kernel ();
}
```

```
...
void MT_sem_init (struct sema *sem, int init_count)
{
  enter_MT_kernel ();

  sem->cnt = init_count;
  sem->wait = NULL;

  exit_MT_kernel ();
}
...
void MT_exit (int status)
{
  enter_MT_kernel ();

  current->state = TERMINATED;
  current->ret_code = status;

  while (1) {
    struct tcb *t;
    t = dequeue_head (&current->wait_q);
    if (!t) break;
    wake_up (t);
  }

  current = NULL;
  dispatch ();
}
...
static void init_tcb (struct tcb *t)
{
  t->tid = get_new_tid ();
  t->wait_q = NULL;
  t->magic = 0xdeadbeef;
  memset (&t->context, 0, sizeof (ucontext_t));
  t->prev = t->next = NULL;
}
...
int MT_usleep (int us)
{
  if (us < 0) return -1;

  enter_MT_kernel ();

  current->wake_time = get_real_time () + us;
  add_tail (&timer_q, current);
  block_me ();

  exit_MT_kernel ();
  return 0;
}
```

```
...
int MT_join (int tid, int *result)
{
  struct tcb *t, **ot;

  enter_MT_kernel ();

  t = all_threads;
  ot = &all_threads;

  ... /* set t to thread with id tid */

  if (t->state != TERMINATED) {
    add_tail (&t->wait_q, current);
    block_me ();
  }

  if (result)
    *result = t->ret_code;
  *ot = t->all_next;

  ... /* free t */

  exit_MT_kernel ();
  return 0;
}
...
int MT_set_share (int share)
{
  if (share <= 0 || share > 10000)
    return -1;
  enter_MT_kernel ();

  current->share = share;

  exit_MT_kernel ();

  return 0;
}
```

File system format from HW6:

Disk blocks are 512 bytes. A valid filesystem must contain at least three blocks. The maximum size of a filesystem is the number of blocks that can be described by a signed 32-bit integer (i.e. $2^{31} - 1$). The first block on the disk (called the zero block from here on) is special — it contains the directory. Its format is as follows, working on the assumption that the zero block has been read into a C array declared as `int zero_block[128]`:

zero_block[0]	int magic1 == 0xabbaabba;
zero_block[1]	int magic2 == 0xf00bf00b;
zero_block[2]	int block_size == 512;
zero_block[3]	int block_cnt;
zero_block[4]	int free_list_head;
zero_block[5]	int file0;
zero_block[6]	int file1;
zero_block[7]	int file2;
...	
zero_block[127]	int file122;

The two magic numbers are used to implement a crude validity check on MT filesystems: the filesystem code should abort with an error if they do not have the specified values. For purposes of this assignment, `block_size` is always set to 512. `block_cnt` stores the number of blocks that the filesystem contains and `free_list_head` contains the block number of the first block on the free list. The first 4 bytes of each free block are required to store an integer pointing to the next block on the list, with -1 indicating the end-of-list.

Each of the 123 file entries either contains -1 for a free slot, or the disk block number of the inode of a file in the filesystem. An inode is a disk block containing the meta-data associated with a single file. Its format is as follows, working on the assumption that the zero block has been read into a C array declared as `int inode[128]`:

inode[0]	int magic == 0xdeadbeef;
inode[1–16]	char filename[64];
inode[17]	int size;
inode[18]	int block0;
inode[19]	int block1;
inode[20]	int block2;
...	
inode[127]	int block109;

The magic number is used to implement a crude validity check on MT filesystems: the filesystem code should abort with an error if an inode is found to not contain the magic value. The `filename` is an array of 64 bytes that contains up to 63 characters of filename plus a terminating null byte. `size` indicates the size of the file in bytes. Finally, the block pointers describe the disk block numbers of data blocks allocated to the file. It is always the case that block 0 stores bytes 0–511 of a file, block 1 stores bytes 512–1023, etc. It can easily be computed that the maximum size of a file in the MT filesystem is about 56 KB (the digital camera in question apparently supports cutting-edge image compression...).

Every disk block must belong to exactly one of the four categories described above: the zero block, free blocks, inodes, and data blocks. The MT block device driver performs a filesystem validity check at startup time to ensure that this is the case.