CS560 Midterm Exam – February 25, 2010 – James S. Plank

Answer all questions.
Please answer on separate paper – do not put your answers ont the test.
Don't forget to put your name and email address on your answer sheets.

Question 1

In the cbthreads library, a thread has six potential states: RUNNING, READY, BLOCKED, SLEEPING, ZOMBIE and JOINING. On the answer sheet, I have started a diagram that has an ellipse for each state. What I want you do is draw arrows between states which represent all possible transitions between the states. You should label each arrow with the procedure from the cbthread library that causes the transition, and the reason for the transition. I have put two arrows in for you.

Question 2

Suppose we are implementing the Dining Philosophers problem in pthreads, and suppose we have the following procedures available to us:

`int are_my_sticks_available(int id)` – returns whether philosopher id's chopsticks are available.
`int before(int id)` – returns (id + nphil – 1) % nphil
`int after(int id)` – returns(id + 1) % nphil
`double hunger_time(int id)` – returns the current time plus one if philospher id is not hungry. Otherwise it returns the time when the philosopher became hungry.
`void pick_up_stick(int stick)` – picks up the given chopstick. Returns when the chopstick has been picked up.
`void put_down_stick(int stick)` – puts down the given chopstick. Returns when the chopstick is down.
`pthread_mutex_t *lock()` – returns a mutex that all philosophers share.
`pthread_cond_t *cond(int id)` – returns a condition variable for a philosopher.
`Dllist Q` – This is a global variable that is a queue shared by all philosophers.

Below are three implementations of `i_am_hungry()` and one implementation of `i_am_sated()`. Since this is a pthreads implementation, they simply return when they are done (there are no continuations).

```
void i_am_sated(int id)
{
  put_down_stick(id);
  pthread_mutex_lock(lock());
  pthread_cond_signal(cond(before(id)));
  pthread_mutex_unlock(lock());
  put_down_stick(after(id));
  pthread_mutex_lock(lock());
  pthread_cond_signal(cond(after(id)));
  pthread_mutex_unlock(lock());
}
```

```
void i_am_hungry_1(int id)
{
  dll_append(Q, new_jval_i(id));
  pthread_mutex_lock(lock());
  while (!are_my_sticks_available(id) ||
      Q->flink->val.i != id) {
    pthread_cond_wait(cond(id), lock());
  }
  dll_delete_node(Q->flink);
  pthread_mutex_unlock(lock());
  pick_up_stick(id);
  pick_up_stick(after(id));

}
```

```
void i_am_hungry_2(int id)
{
  pthread_mutex_lock(lock());
  while (!are_my_sticks_available(id) ||
      hunger_time(before(id)) < hunger_time(id) ||
      hunger_time(after(id)) < hunger_time(id)) {
    pthread_cond_wait(cond(id), lock());
  }
  pthread_mutex_unlock(lock());
  pick_up_stick(id);
  pick_up_stick(after(id));
}
```

```
void i_am_hungry_3(int id)
{
  pthread_mutex_lock(lock());
  while (!are_my_sticks_available(id)) {
    pthread_cond_wait(cond(id), lock());
  }
  pthread_mutex_unlock(lock());
  pick_up_stick(id);
  pick_up_stick(after(id));
}
```

For each of the `i_am_hungry` implementations, explain how the implementation works, and tell me the pros and cons of the implementation. Concentrate on all the metrics that we talked about in class.

Question 3

The following Unix /bin/sh command runs two processes: "cat" and "grep xxx." Standard input of the cat process comes from the file "cin.txt". Standard output of the cat process is piped to standard input of the grep process. Standard error of the cat process goes to the file cerr.txt, and standard output of the grep process goes to the file gout.txt.

```
UNIX> /bin/sh
$ cat < cin.txt 2> cerr.txt | grep xxx > gout.txt
```

Write a program that performs the exact same mechanics of the Unix command. It should create two processes, hook their inputs and outputs up correctly, and only return when the processes are done. You may use the following system calls:

```
      int close(int d);
      int dup2(int oldd, int newd);
      int execlp(const char *file, const char *arg, ... /*, (char *)0 */);
      int open(const char *path, int flags, mode_t mode);
      int pipe(int *fildes);
    pid_t fork(void);
    pid_t wait(int *status);
  ssize_t read(int d, void *buf, size_t nbytes);
  ssize_t write(int d, void *buf, size_t nbytes);
```

Question 4

A barrier is a synchronization primitive that allows a collection of threads to block, and not to proceed until all threads in the collection have blocked. We can implement them in pthreads to have the following prototypes and semantics:

```
typedef void *barrier;
```

`barrier barrier_create(int nthreads)` – create a barrier. Nthreads is the number of threads that will participate in the barrier.

`void barrier_block(barrier b, int i)` – when a thread calls this, it should block until nthreads have blocked on the barrier with integer i. After nthreads have called barrier_block(b, i), all of the threads should be unblocked.

For example, the following program is guaranteed to print "000001111122222\n":

```
barrier b;

void *thread(void *arg)
{
  int i;

  for (i = 0; i < 3; i++) {
    printf("%d", i);
    barrier_block(b, i);
  }
  if (arg == NULL) printf("\n");
  return NULL;
}
```

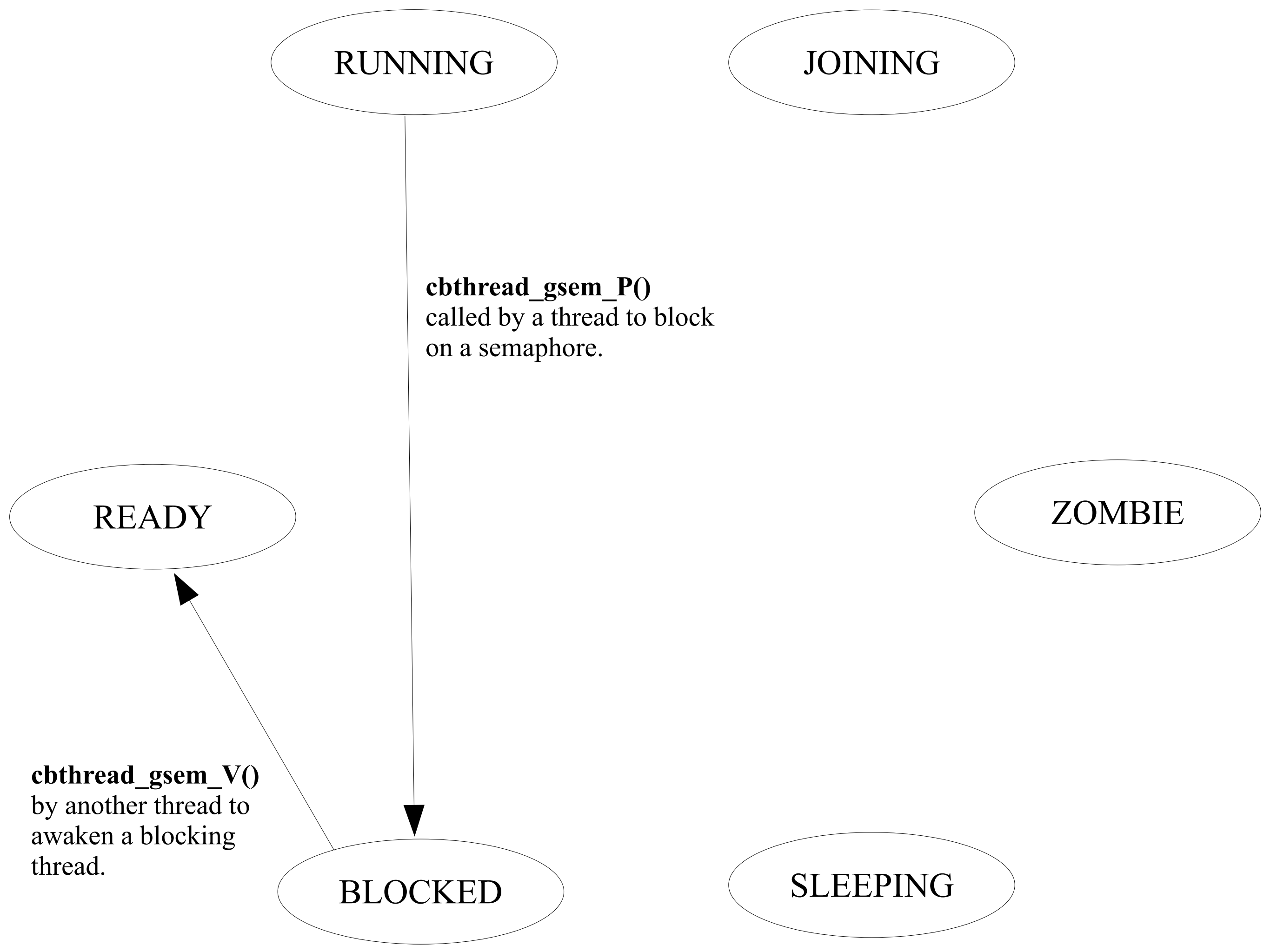
```
main()
{
  pthread_t tids[5];
  void *not_null, *arg;

  not_null = (void *) malloc(1);

  b = barrier_create(5);
  for (i = 0; i < 5; i++) {
    arg = (ij == 0) ? NULL : not_null;
    pthread_create(tids+i, NULL, thread, arg);
  }
  pthread_detach();
}
```

Your job is to write barrier_create() and barrier_block(). Try to be as unrestrictive as possible (for example, perhaps one set of *n* threads calls barrier_block(b, 1) and another set calls barrier_block(b, 2). It should work). I have included prototypes of pthreads procedure calls on the next page.

Answer sheet for Question 1



Prototypes for pthreads:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```