CS 4410 Operating Systems Prelim I, Fall 2015
Profs. Bracy and Van Renesse

NAME: ____ **NetID:__________**

- **This is a closed book examination. You have 120 minutes. No electronic devices of any kind are allowed.**
- You get 1 point for filling in your name and NETID above. You get another point if you fill in your NETID on each (odd-numbered) page. You get 0 points for the whole exam if you don't do any of these…
- **For Coding Questions:** You may use pseudo-code. Descriptions written mostly in English will get no credit. In addition to correctness, elegance and clarity are important criteria for coding questions.
- Show your incomplete work for partial credit. Make any other assumptions as necessary and document them. Brevity is key.
- Please write your solutions within the provided boxes as much as possible. Write clearly. Use the scratch paper at the end if you need to practice your answer first.

Question	Points Possible
0: Name & NetID	2
1: Multiple Choice	14
2: Multicore Synchronization	15
3: Scheduling	20
4:Threads and Processes	14
5: Synchronization	20
6: Deadlocks	15
Total	100

[14pts] 1. Multiple Choice Questions

[2] a) Which of the following is the best way to wait for two predicates to be true?

Your Answer: **A**

(A)
```
with lock:
  while not condA or not condB:
    if not condA:
      condA_cv.wait()
    if not condB:
      condB_cv.wait()
```

(B)
```
with lock:
  while not condA:
      condA_cv.wait()
  while not condB:
      condB_cv.wait()
```

(C)
```
with lock:
  while not condA or not condB:
     condA_cv.wait()
     condB_cv.wait()
```

(D)
```
with lock:
  if not condA or not condB:
    while not condA:
      condA_cv.wait()
    while not condB:
      condB_cv.wait()
```

Check the 12 Commandments

[2] b) Which of the following are (virtually) shared by threads within a single process? In other words, is it instantiated per process instead of per thread? Select all that apply.

(A) Heap
(B) Stack
(C) Code / Program Text
(D) Registers

Your Answer: **A,C**

[2] c) Which of the following operations require the executing code to be operating with high privilege? Select all that apply.

(A) Implementing a monitor
(B) Performing a semaphore P operation
(C) Accessing the device registers of an I/O device, e.g. the disk, keyboard, or network card
(D) Disabling interrupts
(E) Making a system call

Your Answer: **C,D**

NETID:________

[2] d) Which of the following statements about systems calls are **true**?
Select all that apply.
(A) A system call can be caused by events external to the CPU.
(B) All registers must be saved when performing a system call.
(C) The old privilege mode must be saved when performing a system call.
(D) The privilege mode must be changed to "kernel mode" when performing a system call.
(E) The system call handler can run on the stack of the user process that issued the call.

Your Answer: **C,D**

[2] e) You are using a semaphore package which provides 3 functions: `init()`, `P()`, and `V()`.
Which of the following changes to the package could affect the correctness of your code?
Select all that apply.
(A) `P` is modified so that it busy-waits instead of yielding when a resource isn't available.
(B) `init` is modified so that it only accepts 0 or 1 as an initial value.
(C) The implementation stores the count in an unsigned int instead of a signed int.
(D) `V` is modified so that it wakes the thread that most recently called `P`.
(E) Asserts are removed from all three functions.

Your Answer: **B**

Recall: implementation should not affect abstraction!!

[2] f) Which of the following statements about deadlocking is **true**?
Select all that apply.
(A) If a system is not deadlocked at time T, it can always avoid being deadlocked at time T+1.
(B) If a system is in a safe state at time T, it can always avoid being deadlocked at time T+1.
(C) When reducing a Resource Allocation Graph, you should begin with the process which currently holds the most resources.
(D) Modern operating systems use the Banker's Algorithm to determine whether it is safe to give a particular process a particular resource.

Your Answer: **B**

[2] g) Which of the following statements about threads is **false**?
Select all that apply.
(A) Multi-threading is only useful on a multi-processor.
(B) Multi-threading is only useful when a task can be parallelized.
(C) There are performance benefits to running threads of the same process one after the other on the same processor.
(D) Multi-threading requires operating system support for managing multiple PCBs.

Your Answer: **A,D**

[15pts] 2. Multicore Synchronization using Compare-and-Swap

For this question you may assume reading and writing individual memory locations of size 'int' are atomic and that sizeof(int) == sizeof(int *), *i.e.,* pointers and integers have the same size. Also, the "C"-like language below is really pseudo-code. Don't worry about casting pointers to integers or vice versa. A memory location is a memory location! Also, in the solutions you are allowed to *busy-wait*.

Many CPU architectures provide a Compare-And-Swap (CAS) instruction with the following semantics (here *addr* is the address of some memory location). Note that (not counting the fetch of the instruction itself, which is hopefully from the cache) CAS involves a single read and at most a single write cycle on the memory bus (reading and writing *addr):

```
ATOMIC bool CAS(int *addr, int oldval, int newval) {
        if (*addr != oldval)
                return FALSE;
        *addr = newval;
        return TRUE;
}
```

a) [3] Implement a Test-And-Set function using CAS that does (not counting fetching instructions) a single memory bus read and at most a single bus write cycle:

```
bool TAS(int *addr) {

    return !CAS(addr, 0, 1);

}
```

TAS returns 0 if the lock is successfully grabbed, whereas CAS returns 1 upon success, hence the '!' (negation)

b) [3] Using CAS, implement an atomic increment operation: INC(&x) increments *x* by 1. In the absence of contention (*i.e.,* no multiple threads trying to increment *addr at the same time), your code should, at most, read memory twice and write memory once (*i.e.,* at most three memory bus accesses). You can assume that local function variables are kept in CPU registers.

```
void INC(int *addr) {

    register int oldval = *addr;
    while (!CAS(addr, oldval, oldval + 1))
        oldval = *addr;

    // Loop required in case multiple threads try to update *addr at the same time

}
```

c) [3] Use CAS to complete the implementation of an append-only *lock-free* list (*i.e.*, you are not to declare a separate variable for a spinlock). The list is maintained in reverse order.

```
struct item {
        struct item *prev;     // points to previous item added to the list (null for first item)
        int value;             // contains the value in this entry
};
struct item *list = NULL;      // points to last item added to the list (null if list is empty)

void add(int val) {            // add value to the list
        struct item *node = malloc(sizeof(struct item));
        node->value = val;
        node->prev = list;     //replace these 2 lines
        list = node;           //with thread-safe code below:
```

(The lines `node->prev = list;` and `list = node;` are struck through on the page.)

```
do
        node->prev = list;
while (!CAS(&list, node->prev, node);
```

```
}
```

d) [3] The following code checks if *val* is in the list. Is it thread-safe? If so, explain what is meant by thread-safety. If not, explain why not. (add() is the only update operation.)

```
bool check(int val){
        struct item *node = list;
        while (node != NULL) {
                if (node->value == val)
                        return TRUE;
                node = node->prev;
        }
        return FALSE;
}
```

Yes. While it is true that check(X) may or may not find X in case add(X) is executed concurrently, both are valid executions. add() appears to execute atomically with respect to check(). A requirement is that check(X) *has to* find X if add(X) completed before invoking check(X), and *should not* find X if add(X) is invoked after check(X) completed.

e) [3] Describe advantages of CAS over TAS. Consider, for example, how hard it would be to implement a lock-free counter or list data structure using either CAS or TAS:

CAS can easily emulate TAS, but CAS makes it possible to build lock-free counters and data structures. Solutions based on CAS often require fewer memory accesses and less memory.

[20pts] 3. A LUF-ly Scheduling Algorithm

A creative Cornell student with a strong sense of Ithaca-style fairness came up with a new scheduling policy: Least Used First (LUF). When given the choice to schedule two processes (jobs) on the run queue, the scheduler will select the one that has used the fewest CPU cycles thus far. In case of a tie, the queue is otherwise FIFO. When a process is de-scheduled it goes to the end of the queue. *The CPU should not site idle when there are processes on the run queue.*

Recall that a job has an arrival time and a duration (the amount of CPU time it will need, in time units). The turnaround time is the time between arrival and the time the job finishes. The response time is the time between arrival and the time the job is first scheduled. The waiting time is the total amount of time the job spent on the run queue waiting to be scheduled (*i.e.,* for CPU-bound jobs it is turnaround time – duration time). Assume all jobs are purely CPU-bound.

[8] a) Supposing a running job is only pre-empted when a new job arrives (no interrupts), fill the following table:

Job	Arrival Time	Duration	Turnaround Time	Response Time	Waiting Time
A	0	25	55	0	30
B	15	25	45	0	20
C	25	5	5	0	0
D	40	5	5	0	0

[8] b) Suppose the clock interrupts the CPU every 2 time units (*i.e.,* at times 0, 2, 4, etc.) and jobs arrivals do not preempt the current job. If a job arrives at the same time as a clock interrupt, the job can run immediately. Fill in the following:

Job	Arrival Time	Duration	Turnaround Time	Response Time	Waiting Time
A	0	25	59	0	34
B	15	25	45	1	20
C	25	5	6	1	1
D	40	5	5	0	0

[4] c) Can the LUF policy cause starvation? Explain your answer.

Yes. A process that has been running for a long time will be starved from CPU cycles if a series of new processes arrives that only run for a short amount of time.

NETID:________

[14pts] 4. What does it do???

[7] a) Look at the C program below. Write its output in the box to the right. FYI: **waitpid(pid, &status, 0)** waits for process pid to finish. System calls are underlined---you can assume there won't be errors. #include's have been left out.

```
char array[] = "qwertyuiopasdfghjklzxcvbnm";
int len = sizeof(array);     // 26 initially

void mystery(){
    char first = array[0];
    int status, i;
    int j = 0;
    int pid = fork();
    if (pid == 0) {
        for (i = 1; i < len; i++)
            if (array[i] < first)
                array[j++] = array[i];
    } else {
        waitpid(pid, &status, 0);  // wait for pid
        write(1, &first, 1);           // prints first
        for (i = 1; i < len; i++)
            if (array[i] > first)
                array[j++] = array[i];
    }
    len = j;
    if (len > 0)
        mystery();
}

int main(){
    mystery();
    return 0;
}
```

abcdefghijklmnopqrstuvwxyz

Run it yourself if you don't believe it. fork() is used here to copy the array before splitting it. The child retains all the chars that are before 'first'. The parent waits for the child, prints 'first', and retains the chars that are after 'first'. This process is repeated recursively (tail recursion---can be converted to loop trivially) until the array is empty.

This kind of "fork and join" pattern is a common concurrency technique, although in this particular implementation fork() is used for copying and not for concurrency.

[7] b) Describe briefly how *fork*() works. Mention PCBs, code, data, and stack segments, CPU registers, fork's return value, and the run queue. Use active voice ("program is run" = passive, "user runs a program" = active).

When a process forks, the kernel allocates a new PCB and memory for the new child process and copies the PCB, segments, and registers of the parent. The kernel assigns a new process id to the child process, and sets its return value register to 0. The kernel sets the parent's return value register to the child's process id. The kernel places both processes on the run queue.

[20pts] 5. Pooling Resources

You are to simulate a pool hall with *N* pool tables numbered 0 to *N* - 1. Players (threads) that arrive at the pool hall are looking for tables that have one (preferred) or no players at them. Once there are two players at a table, they play a game of pool and then they leave (separately). When *both* players have left the pool table, the table becomes available for new players. Fill in the missing code below using Python-ish pseudo-code. Make sure to distinguish class variables from local function variables, and don't add global ones. Use Mesa locks and condition variables. No busy waiting!

```
N = 8   # number of pool tables
class PoolHall:
  def __init__(self):                    add additional class variables below if you need them
    self.lock = Lock()                          # monitor lock
    self.nAtTable = [0 for tno in range(N)]     # number of players at each table (at most 2 players)
```

```
self.nLeft = [0 for tno in range(N)]
self.tableCond = [Condition(self.lock) for tno in range(N)]
self.availCond = Condition(self.lock)
```

```
  def player_enters(self):     # wait until there's a table, and then wait for there to be two players at the table. Return
                               # the table number (index into self.nAtTable)
    with self.lock:                              insert your code below
```

```
while self.table_avail() < 0:          # wait for a table
    self.availCond.wait()
tno = self.table_avail()               # which table is available?
self.nAtTable[tno] += 1                # occupy the table
if self.nAtTable[tno] == 1:            # am I the only one?
   while self.nAtTable[tno] != 2:      # wait for another player
       self.tableCond[tno].wait()
else:                                  # table full
   assert self.nAtTable[tno] == 2
   self.tableCond[tno].notify()        # notify the waiting player
   self.nLeft[tno] = 0                 # 0 players that have left
return tno                             # return the table number
```

NETID:________

class PoolHall *continued*:

```
  def player_exits(self, tno):  # Release table tno.  Once both players have left, the table is available again.  The players
                                # don't have to wait for one another to leave
    with self.lock:                         insert your code below
```

```
      assert self.nAtTable[tno] == 2 and self.nLeft[tno] < 2
      self.nLeft[tno] += 1              # incr. #players leaving
      if self.nLeft[tno] == 2:          # all players left
        self.nAtTable[tno] = 0          # make table available
        self.availCond.notifyAll() # notify players waiting
```

```
  # Here's a handy function you may like to use.  Returns a table number tno if that table is available.  Prefers
  # tables with one player waiting over empty tables.  Returns -1 if all tables are taken.
  def table_avail(self):
    for tno in range(N): if self.nAtTable[tno] == 1: return tno
    for tno in range(N): if self.nAtTable[tno] == 0: return tno
    return -1
```

```
class Player(Thread):
  def __init__(self, poolhall, pid):
    Thread.__init__(self)
    self.poolhall = poolhall
    self.pid = pid
```

```
  def run(self):
      while True:
        tno = poolhall.player_enters()
        print "Player ", self.pid, " got table ", tno
        # PLAY SOME POOL
        poolhall.player_exits(tno)
```

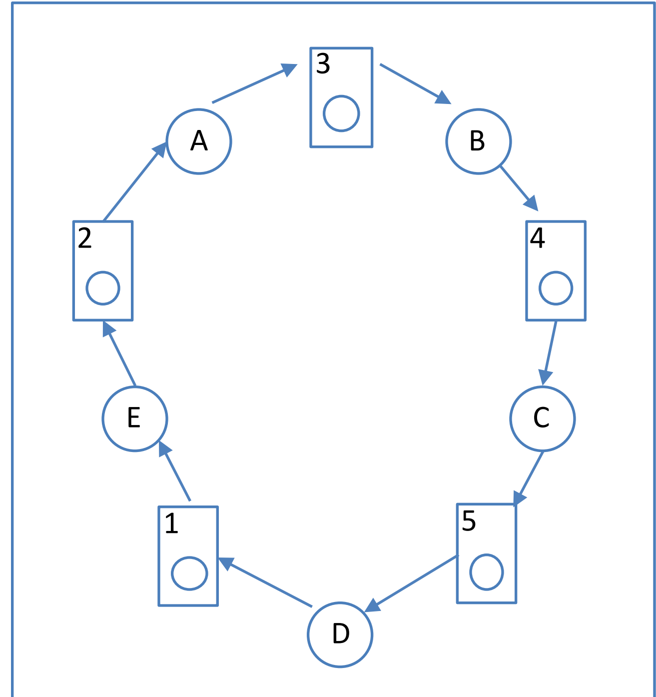
```
poolhall = PoolHall()
for pid in range(NPLAYERS):
  Player(poolhall, pid).start() # pid = player identifier
```

if you like you can leave some explanation of your code below

[15pts] 6. Variations on Dining Philosophers

Recall the five Dining Philosophers (and five chopsticks) sitting around a circular table: if they all pick up their right chopstick before their left, they can end up in a deadlocked situation.

a) [4] In the box to the right, draw a Resource Allocation Graph showing the Philosophers and chopsticks and a deadlocked situation.



b) [3] Assume L philosophers are left-handed and pick up their left chopstick first, while the remaining R philosophers are right-handed and pick up their right chopstick first. Fill out the following table, indicating *yes*, *no*, or *depends* (describe scenario briefly in the last case).

L	R	Deadlock Possible?
5	0	YES (each philosopher picks up left chopstick first)
4	1	NO (symmetry broken)
3	2	NO (ditto)

c) [3] The five philosophers instead decide to place N chopsticks in a heap in the middle of the table. Philosophers can now pick up one chopstick at a time, but still need two to eat. Say if deadlock is possible (YES, NO, DEPENDS) in each of the following cases:

N	Deadlock Possible?
<5	YES (each philosopher picks up 0 or 1 chopstick until nothing is left)
5	YES (each philosopher picks up 1 chopstick until nothing is left)
>5	NO (always at least one philosopher who can eat)

d) [5] In a recent NASA discovery, planet Mars has three-handed philosophers. Consider a table with five three-handed philosophers and a pile of N chopsticks in the middle of the table. Each philosopher needs 3 chopsticks to eat. What is the smallest N such that deadlock is impossible?

11

With 10 chopsticks, each philosopher could pick up two chopsticks and end up deadlocked. With 11, there is always at least one philosopher who can eat.