OS161 exercises (from old exams)

NOTE: the text includes comments on exam correction criteria and/or possible/frequent errors (they were deliberately left)

1. Function `as_define_region` (file `dumbvm.c`) is partially shown in the figure below. Suppose that parameters `as` and `vaddr` reveive (hexadecimal) values `0x80048720` and `0x412370`, respectively, and `sz` value (decimal) `4128`. PAGE_SIZE is defined as `4096`, and PAGE_FRAME as `0xfffff000`. Simulate the instructions shown by showing, for each of them, (hexadecimal) values of operand(s) and result (the value assigned to a variable).

```
as_define_region(struct addrspace *as, vaddr_t vaddr, size_t sz,
         int readable, int writeable, int executable) {
    size_t npages;
    /* Align the region. First, the base... */
    sz += vaddr & ~(vaddr_t)PAGE_FRAME;
    vaddr &= PAGE_FRAME;
    /* ...and now the length. */
    sz = (sz + PAGE_SIZE - 1) & PAGE_FRAME;
    npages = sz / PAGE_SIZE;
    ...
}
```

	4096 = 0x1000 4128 = 0x1020
`sz += vaddr & ~(vaddr_t)PAGE_FRAME;`	0x1020 += 0x412370 & 0x000FFF sz <- 0x1020+0x370 = **0x1390**
`vaddr &= PAGE_FRAME;`	0x412370 &= 0xFFF000 vaddr <- **0x412000**
`sz = (sz + PAGE_SIZE - 1) & PAGE_FRAME;`	sz <- (0x1390+0xFFF) & 0xFFF000 = **0x2000**
`npages = sz / PAGE_SIZE;`	npages <- 0x2000 / 0x1000 = 2

Now suppose that the value received by `sz` be smaller than the page size (e.g. `4090`). Is it possible to obtain value 2 for `npages`? (motivate the answer)

Yes. It is indeed possible, as the segment is aligned to a page multiple/boundary, both for its start and end positions. As a matter of fact, we have internal fragmentation at both sides of the segment (first and last page). With value 4090 = 0xFFA, `sz` would first get value `0xFFA+0x370 = 0x126A`, then `0x2000`.

2. The figure below shows function `getfreeppages`, that allocates an interval of `npages` contiguous free pages in physical memory.

```
static paddr_t
getfreeppages(unsigned long npages) {
  paddr_t addr = 0;
  long i, first, found, np = (long)npages;

  if (!isTableActive()) return 0;
  spinlock_acquire(&freemem_lock);
  for (i=0,first=found=-1;i<nRamFrames;i++) {
    if (freeRamFrames[i]) {
      if (i==0 || !freeRamFrames[i-1])
        /* set first free in an interval */
        first = i;
      if (i-first+1 >= np)
        found = first;
    }
  }
  if (found>=0) {
    for (i=found; i<found+np; i++) {
      freeRamFrames[i] = (unsigned char)0;
    }
    allocSize[found] = np;
    addr = (paddr_t) found*PAGE_SIZE;
  }
  spinlock_release(&freemem_lock);
  return addr;
}
```

/* first-fit */

/* best-fit */

Which allocation policy is implemented by the function? Choose among best-fit, worst-fit, first-fit or other (motivate the answer) ?

None of the three proposed.
First-fit NO because iterations don't stop with the first solution found
Best-fit and Worst fit NO as no search for minimum/maximum is implemented/operated
The returned solution is the last one among those that are (possibly) found. One could name the policy "last-fit".

FREQUENT/POSSIBLE ERROR: *consider the proposed solution as implementing a* ***first-fit*** *policy*
COMMENT (for the next question, modifying the program): *most of the students that were able to notice the last-fit policy were also able to correct it and transform it to first-fit.*
Fewer students were able to face the search problem: finding the ***"minimum among the intervals of length >= np"*** *(or npages).*

Notice that the proposed algorithm is O(nRamFrames) and that the original/given code checks the interval length (for sake of simplicity) at each intermediate entry within an interval.
If worst-fit (searching a maximum) was the goal, the patch could be simpler: just checking the length at each iteration and possibly updating a max variable.
On the other hand, best-fit needs to check against the minimum just at the end of an interval (not in the middle)
*I've seen solutions with complexity O(nRamFrames*npages) based on a double iteration; though ness efficient, they can be considered as correct, whereas I don't consider as acceptable any solution based on an "existing" auxiliary array filled with interval lengths: such an array would partly solve the problem, but (beyond the memory cost) it would require an extra cost in order to keep it up-to-date at each allocation/freeing (or one should show how to fill the array when needed, as a preprocessing step of* `getfreeppages`*).*

Modify the function (filling the empty frame with your modifications/patches, limited to instructions in grey background) in order to implement a first-fit policy, as well as a best-fit one.

```
static paddr_t
getfreeppages(unsigned long npages) {
  paddr_t addr = 0;
  long i, first, found, np = (long)npages;

  if (!isTableActive()) return 0;
  spinlock_acquire(&freemem_lock);
  for (i=0,first=found=-1;i<nRamFrames;i++) {
    if (freeRamFrames[i]) {
      /* check every free frame (even
         intermediate ones. If it is the
         first of an interval, "remember"
         the start index. */
  if (i==0 || !freeRamFrames[i-1])
        /* set first free in an interval */
        first = i;
      /* any interval compatible with np
         is checked. So the last one is
         returned */
  if (i-first+1 >= np)
        found = first;
    }
  }
  if (found>=0) {
    for (i=found; i<found+np; i++) {
      freeRamFrames[i] = (unsigned char)0;
    }
    allocSize[found] = np;
    addr = (paddr_t) found*PAGE_SIZE;
  }
  spinlock_release(&freemem_lock);
  return addr;
}
```

```
/* WARNING: this is just one correct solution:
   other (similar) ones are possible */

/* first-fit: stop iterations when solution
   found */
/* first version: loop controlled by flag */
...
   for (i=0,first=found=-1;
       i<nRamFrames && found<0; i++) {
...
/* second variant: non structured loop */
...
      if (i-first+1 >= np) {
        found = first;
        break;
      }
...

/* best fit: search the minimum (by chacking
length just at the end of a free interval) */
int min;
...
   /* multiple nested if statements could be
      replaced by a single if with a condition
      obtained by AND-ing all sub-conditions
    */
   /* if last frame in an interval */
   if (i==nRamFrames-1 || !freeRamFrames[i+1])
     /* if size OK */
     if (i-first+1 >= np)
       /* if better than temporary best(min) */
       if (found<0 || i-first+1 < min) {
        found = first; min = i-first+1;
       }
...
```

3. Explain, within the framework of function `syscall()`, the role of variable `callno`, that is assigned value `tf->tf_v0`.

The variable works as the selector of the system call to be executed, used in function `syscall` in order the drive the `switch-case` statement. The `tf->tf_v0` field in the trapframe is assigned by the (assembly) instructions that handle the trap and call `mips_trap` and `syscall`.

Discuss/comment the following instructions, located at the end of function `syscall()` (specify the meaning/content of the `tf_v0` e `tf_a3` within the trapframe)

```
if (err) {
  tf->tf_v0 = err;
  tf->tf_a3 = 1;
}
else {
  tf->tf_v0 = retval;
  tf->tf_a3 = 0;
}
```

The instructions handle the return value status:

- `v0` holds the return value of the system call (equal to the error code in case of error)
- `a3` holds the error status success(0)/error(1)

4. Consider the implementation of a lock in OS161. Which thread should be considered as the owner of the lock?
   - The thread that created the lock?
     NO. Creating the lock doesn't mean to be the owner.
   - The last thread calling function `lock_acquire`?
     NOT necessarily. Just in case the lock_acquire was done on the lock that we are considering (so not another one) and the thread actually acquired the lock (so not still waiting)
   - other…(complete)
     See previous answer: the owner of a lock is the thread that called `lock_acquire` on that lock and passed the possible wait (for the lock to become available).

Consider the implementation of functions `lock_release` and `lock_do_i_hold` shown in the figure below. The function contain errors: identify them and provide a possible correction/patch (motivation/explanation needed)

```
void loc_release(struct lock *lock) {
  KASSERT(lock != NULL);
  spinlock_acquire(&lock->lk_lock);
  KASSERT(lock_do_i_hold(lock));
  lock->lk_owner=NULL;
  wchan_wakeone(lock->lk_wchan, &lock->lk_lock);
  spinlock_release(&lock->lk_lock);
}
```

```
bool
lock_do_i_hold(struct lock *lock) {
  spinlock_acquire(&lock->lk_lock);
  if (lock->lk_owner==curthread)
    return true;
  spinlock_release(&lock->lk_lock);
  return false;
}
```

lock_do_i_hold is wrongly handling the "return true" without releasing the spinlock. A possible correct version of the function can be obtained by usong a Boolean flag (ret) and a unified return statement (as an alternative, one could explicitly add spinlock_release before returning true):

```
bool lock_do_i_hold(struct lock *lock) {
  bool ret;
  spinlock_acquire(&lock->lk_lock);
  ret = lock->lk_owner==curthread;
  spinlock_release(&lock->lk_lock);
  return ret;
}
```

`lock_release` is wrongly acquiring the spinlock before calling `lock_do_i_hold`, that will internally try to acquire the same spinlock. So this would be a deadlock problem. Moving `spinlock_acquire` after calling `lock_do_i_hold` is a possible solution.

5. Consider a multi-core system (sys161 configured with multiple CPUs). Explain why mutual exclusion cannot be guaranteed by simple interrupt disabling/enabling.

Because the interrupt would be disabled just on the current CPU, which wouldn't prevent the execution (and interruption) of other threads on other CPUs.
WARNING: *Extending interrupt disabling to all CPUs would not be enough, as parallel threads could be already running on them, potentially competing for shared resources. So interrupt disabling only works on a single core platform, as the running thread is guaranteed to be the only one running at a given time.*

Given the code below (reduced to essential parts) for semaphore functions P and V

```
void P(struct semaphore *sem) {
   spinlock_acquire(&sem->sem_lock);
   while (sem->sem_count == 0) {
     wchan_sleep(sem->sem_wchan,
                 &sem->sem_lock);
   }
   sem->sem_count--;
   spinlock_release(&sem->sem_lock);
}
```

```
void V(struct semaphore *sem) {
   spinlock_acquire(&sem->sem_lock);
   sem->sem_count++;
   KASSERT(sem->sem_count > 0);
   wchan_wakeone(sem->sem_wchan,
                 &sem->sem_lock);
   spinlock_release(&sem->sem_lock);
}
```

Answer the following questions:

- What is the role of the spinlock (in both functions)?

The spinlock is needed in order to allow using the wait-channel, and call functions `wchan_sleep` and `wchan_wakeone` an owned spinlock. The spinlock is mandatory. In practice, it has the role of providing mutual exclusion on `sem->sem_count`
WARNING: *A "generic" answer, just explaining the role of spinlocks is considered just as a partially correct answer. The question is on the role of the spinlock within the given functions, not in general.*

- Why does P wait on a `while` loop, instead of just adopting an `if (sem->sem_count == 0)`, conditional statement, whereas no loop is present in the V function?

Because the synchronization scheme implemented between `wchan_wakeone` and `wchan_sleep` (waking up the waiting thread, implemented following the "Mesa" semantics, instead of the "Hoare" one) doesn't guarantee that the condition, true when calling `wchan_wakeone`, still be true at the return from `wchan_sleep`. Other threads could modify it in the meanwhile.
FREQUENT ERROR: *consider "spurious"/erroneous awakenings.* ***NO ERRONEOUS AWAKENINGS OCCUR****. The problem simply lies with the presence of multiple concurrent threads/processes that could be correctly working and modifying the observed condition. The threads could be for instance activated by other calls to V, and, without guaranteeing a chronology-based scheduling (Hoare semantics), they could modify the semaphore counter before the thread goes from READY to RUN status.*

- Why does `wchan_sleep` receive a spinlock as parameter? Is it a reason/motivation holding for `wchan_wakeone`?

The `wchan_sleep` function needs to release the spinlock, before putting the thread on "wait" state. The spinlock has to be later acquired upon awakening (before returning to the caller). The `wchan_wakeone` function has nothing to do with the spinlock (the Unix/Linux versions of the function, for instance, have NO spinlock parameter): in OS161 is simply passed in order to be checked (by a KASSERT, to prevent potential errors by the programmer), as the thread is supposed to own the spinlock when calling `wchan_wakeone`.
FREQUENT ERRORS: *consider that the spinlock is necessary to guarantee mutual exclusion within either `wchan_sleep` or `wchan_wakeone`: NO. The functions do not internally need the spinlock as they do not have a critical section. Spinlock ownershiop is asserted in order to verify that the CALLING THREAD operates in a correct way, NOT to guarantee that the wait channel operates correctly (though obviously `wchan_sleep` needs to release and later acquire the spinlock).*
Another (worse) error: think that the thread goes in wait state on the spinlock. This isn't true: the spinlock just provides (fast) mutual exclusion, not wait/signal synchronization

- Can a call to `wchan_wakeone` wake up more than on ***thread*** waiting on `wchan_sleep`? If NO, why? If YES, how can we release just one among the threads waiting on function `P`?

NO. Function `wchan_wakeone` guarantees that just one thread is awakened, among all the ones waiting on `wchan_sleep`. The `wchan_wakeall` function is the one waking all waiting threads.