Department of Electrical Engineering and Computer Science

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.828 Operating System Engineering: Fall 2005

Quiz II Solutions

Average 84, median 83, standard deviation 10.

I O/S Bugs

1. [10 points]: The correctness of JOS depends in part on interrupts not occurring while the kernel is executing. Suppose that by mistake your JOS kernel didn't turn off interrupts when entering the kernel. Explain a specific problem that would arise if a clock interrupt occurred while executing the code for a system call in the kernel.

Answer: The interrupt will execute on the kernel stack, and will push its state below the system call's stack frames. `trap()` in the interrupt will overwrite the current environment's `env_tf` with a return point in the system call code in the kernel. There are two possible sequences of events now, both of which lead to errors.

If the interrupt returns without calling `sched_yield()`, it will return back into the system call. When the system call returns, it will use the same `env_tf`, and will return into the middle of the kernel system call code rather than back to user space.

If the interrupt calls `sched_yield()` to switch to a new environment, it will return to the user space in the new environment. The old environment's system call will never be finished: next time it runs, it will run in user space as if the system call had returned.

II Shells

The file `script` contains the following two lines:

```
echo first
echo 2nd
```

You type the following command to a shell:

```
$ sh < script > out
```

2. **[5 points]:** What should file `out` contain after this command?

Answer:

```
first
2nd
```

The JOS library sets the PTE_SHARE bit in the page table entry of the page that contains each struct Fd. This bit causes spawn and fork to share Fd pages read/write with child environments, instead of copying them.

Here's what struct Fd looks like, from inc/fd.h:

```
struct Fd {
  int fd_dev_id;
  off_t fd_offset;
  int fd_omode;
  union {
    // File server files
    struct FdFile {
      int id;
      struct File file;
    } fd_file;
  }
}
```

3. [10 points]: What would file out contain in the above example if the JOS library copied Fd pages in spawn and fork, instead of sharing them read/write? Explain your answer.

Answer: In unmodified JOS, the page containing the struct Fd is shared between the parent, the children, and the file server. Thus the children see each others' modifications to Fd.fd_offset when they write the file, so the second echo starts writing where the first left off. Similarly, the children share the Fd page with the server, so both children see the server's changes to the Fd.fd_file.file.f_size field when either child sends a set_size RPC to the server to increase the file size.

If the Fd page isn't shared, the file server will see that its Fd page has no other references. This will cause it to believe that all clients have closed the file descriptor, and it will reject both clients' set_size RPCs, leaving the file size at zero.

If the file server didn't check the reference count of the Fd page, it's likely that out would end up containing 2nd followed by a newline. The reason is that the second echo would start writing at offset zero (since it doesn't see the first echo's change to the offset), would think that it had appended bytes to the end of the file (since it doesn't see changes to the size field), and would thus send a set_size RPC to set the file size to four.

III IPC

Jochen Liedtke's paper *Improving IPC by Kernel Design* suggests, in Section 5.3.5, that IPC performance may be improved if the environment sending a message yields the CPU to the receiving environment. It occurs to you that the following simple modification to JOS might achieve the same effect:

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
  struct Env *e;
  int r;

  if((r = envid2env(envid, &e, 0)) != 0)
    return r;

  if(e->env_ipc_recving == 0)
    return -E_IPC_NOT_RECV;

  // we've omitted code to deal with the page mapping...

  e->env_ipc_recving = 0;
  e->env_ipc_from = curenv->env_id;
  e->env_ipc_value = value;
  e->env_status = ENV_RUNNABLE;

  // here's the original code:
  // return pgmap;

  // here's your modification:
  curenv->env_tf.tf_regs.reg_eax = pgmap;
  sched_yield();
}
```

Suppose that most IPCs are sent as part of remote procedure calls (RPCs), in which a client sends a request to a server and the server sends a response. The client environment sends the request message to the server and waits to receive the response; the server then sends the response and waits to receive the next request. You should assume that, at the time the client sends the request, the server is waiting in `sys_ipc_recv()`. As a reminder, the bottom of the page shows `ipc_send()` from `lib/ipc.c` and `sys_ipc_recv()` from `kern/syscall.c`.

4. [10 points]: How would the preceding modification to `sys_ipc_try_send()` affect JOS's IPC performance, where performance is measured as the number of system calls and returns required to perform an RPC? Explain the specific reasons for any change in performance.

Answer: The modification will increase the number of system calls. When the server sends the response, the client will not already be waiting to receive, so the server's first send will fail and it will have to call `sys_yield()`. Without the modification, the client's send returns and the client calls `sys_ipc_recv()` before the server runs.

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
  while(1){
    int r = sys_ipc_try_send(to_env, val, pg, perm);
    if(r != -E_IPC_NOT_RECV)
      return r;
    sys_yield();
  }
}

static int
sys_ipc_recv(void *dstva)
{
  if(dstva < (void*)UTOP && ROUNDDOWN(dstva, PGSIZE) != dstva)
    return -E_INVAL;

  curenv->env_ipc_recving = 1;
  curenv->env_ipc_dstva = dstva;
  curenv->env_ipc_value = 0;
  curenv->env_ipc_from = 0;
  curenv->env_ipc_perm = 0;
  curenv->env_status = ENV_NOT_RUNNABLE;
  sched_yield();
}
```

IV Soft Updates

Recall the soft updates technique described in McKusick and Ganger's *Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast File System*. You run the following benchmark on a soft updates file system:

```
fd = creat("foo", 0666);
for(i = 0; i < 100; i++){
  write(fd, buf, 8192); /* append a block to the file's end */
  fsync(fd); /* force the data to disk */
}
```

You find that each iteration of the for loop causes four disk writes, in this order:

A. part of the bit-map of free blocks,

B. the new data block,

C. one of the i-node's indirect blocks, and

D. the i-node itself.

The file system uses 8192-byte blocks.

5. [5 points]: Why does the file system write the new data block before the indirect block? Give an example of what could go wrong if the file system performed these two writes in the opposite order.

Answer: If a crash and reboot occurs after the indirect block is written but before the data block is written, the file will contain a block of junk (data from some previously deleted file).

6. [10 points]: Why does the file system write the bit-map of free blocks before the indirect block? Give an example of what could go wrong if the write to the bit-map of free blocks occurred after the write to the indirect block.

Answer: If a crash and reboot occurs after the indirect block is written but before the bit-map is written, the block will be both marked free and in the file. Now the block could be allocated to a second file at the same time it is in use by the first file, which is not correct.

You run the same benchmark on an FFS file system (the predecessor to soft updates). The benchmark is faster on an FFS file system, requiring only three disk writes per loop iteration; FFS does not update the bit-map of free blocks on each iteration. Instead, FFS writes any dirty parts of the bit-map once every 30 seconds.

7. [10 points]: Explain why it's OK for FFS to write the free bit-map after it writes the indirect block, even though it is not OK for soft updates.

Answer: FFS's `fsck` re-builds the free bit-map after each reboot, by looking at all files and collecting the set of all blocks not used by any file.

V SMP JOS

Suppose you would like to modify JOS to run on a symmetric multi-processor (SMP): a machine with eight x86 CPUs, a shared bus, and a single memory system. Based on advice you heard in the 6.828 project presentations, you arrange for at most one CPU at a time to execute in the JOS kernel. Any number of CPUs can be executing in user space (for different environments). You ensure at most one CPU in the kernel by acquiring a spin-lock in `trap()`, and releasing it in `env_pop_tf()`:

```
struct Lock big_lock;
struct Env *cpu_env[8]; // the environment running on each CPU

void
trap(struct Trapframe *tf)
{
  acquire(&big_lock);

  curenv = cpu_env[cpu_num()]; // find the environment running on this CPU
  curenv->env_tf = *tf;

  trap_dispatch(&curenv->env_tf);

  if (curenv && curenv->env_status == ENV_RUNNABLE)
    env_run(curenv);
  else
    sched_yield();
}

void
env_pop_tf(struct Trapframe *tf)
{
  release(&big_lock);

  __asm __volatile("movl %0,%%esp\n"
    "\tpopal\n"
    "\tpopl %%es\n"
    "\tpopl %%ds\n"
    "\taddl $0x8,%%esp\n" /* skip tf_trapno and tf_errcode */
    "\tiret"
    : : "g" (tf) : "memory");
  panic("iret failed");  /* mostly to placate the compiler */
}
```

Each CPU has a separate stack on which it executes when it enters the kernel. The `cpu_env` array holds, for each CPU, a pointer to the environment that is running on that CPU, and `cpu_num()` returns the current CPU's number (0 through 7). Each CPU has its own idle environment that `sched_yield()` runs when there is nothing else to do.

Here's the kernel code to allocate a page of physical memory, from `kern/pmap.c`:

```
int
page_alloc(struct Page **pp_store)
{
  *pp_store = LIST_FIRST(&page_free_list);
  if (*pp_store) {
    LIST_REMOVE(*pp_store, pp_link);
    page_initpp(*pp_store);
    return 0;
  }
  return -E_NO_MEM;
}
```

8. [5 points]: Give an example of a way in which `page_alloc()` would likely behave incorrectly if multiple CPUs were allowed to execute in the JOS kernel at the same time.

Answer: If two CPUs call `page_alloc()` at the same time, they might both allocate the same page.

It turns out that the big lock strategy is not quite sufficient to make JOS work correctly on an SMP. Consider, for example, a simple `sched_yield()` algorithm that runs the next environment marked `ENV_RUNNABLE`, and ignores environments marked `ENV_FREE` or `ENV_NOT_RUNNABLE`.

9. [10 points]: Why is this simple scheduler incorrect on an SMP, and how can it be fixed?

Answer: Two CPUs might both decide to run the same environment. There needs to be a new environment state, `ENV_RUNNING`, to indicate environments that are running.

Suppose that your application consists of dozens of environments that constantly send each other IPC messages; there are always more environments waiting to run than there are CPUs. You notice that your applications spend essentially all of their time in the JOS kernel, specifically the system calls that send and receive IPC messages.

10. [10 points]: Compare the likely performance of your application on an eight-CPU SMP with performance on an ordinary single-CPU machine. Briefly explain your answer.

Answer: About the same total performance. Only one CPU is allowed in the kernel at a time, so the other seven CPUs will waste their time waiting for the big lock.

You'd like to make the send and receive system calls higher performance by letting them run in parallel on different CPUs as long as the environments involved are distinct. You modify `trap()` and `env_pop_tf()` to not acquire/release the big lock for send and receive system calls. Instead, you want to add locks inside the send and receive system calls that preserve correctness and allow two environments executing on different CPUs to send concurrently to different environments.

11. [10 points]: Add `acquire()` and `release()` calls to the copies of `sys_ipc_try_send()` and `sys_ipc_recv()` below so that they are correct and allow concurrent sends to different environments. Explain where you declare the `Lock` variables. Briefly explain why your locking scheme allows concurrent sends.

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
  struct Env *e;
  int r;

  if((r = envid2env(envid, &e, 0)) != 0)
    return r;

  acquire(e->env_ipc_lock); // HERE

  if(e->env_ipc_recving == 0){
    release(e->env_ipc_lock); // HERE
    return -E_IPC_NOT_RECV;
  }

  // we've omitted code to deal with the page mapping...

  e->env_ipc_recving = 0;
  e->env_ipc_from = curenv->env_id;
  e->env_ipc_value = value;
  e->env_status = ENV_RUNNABLE;
  release(e->env_ipc_lock); // HERE
  return pgmap;
}

static int
sys_ipc_recv(void *dstva)
{
  acquire(curenv->env_ipc_lock); // HERE
  if (curenv->env_ipc_recving)
    panic("already recving!");

  curenv->env_ipc_recving = 1;
  curenv->env_ipc_dstva = dstva;
  curenv->env_status = ENV_NOT_RUNNABLE;
  release(curenv->env_ipc_lock); // HERE
  sched_yield();
  return 0;
}
```

Answer: Every environment has a separate `env_ipc_lock` that guards its IPC receive variables. This locking scheme allows concurrent sends to different environments because the two sends require different locks.

VI 6.828

We would like to hear your opinions about 6.828, so please answer the following questions. (Any answer, except no answer, will receive full credit!)

12. [1 points]: On a scale of 1 to 10, please rate how much you learned from 6.828 (1 is low, 10 is high).

Answer: We (the staff) learned a lot from you (the students) this semester!

13. [2 points]: If you could change one aspect of 6.828, what would it be?

Answer: We're planning to write our own simple operating system for the x86, in the spirit of UNIX V6 but more modern. We hope to use this new O/S in place of V6. We love V6 but it might be better not to have to spend time on its quirks and on the PDP 11/40.

14. [2 points]: Are there any topics you would like to see added to or removed from the class?

Answer: Perhaps we'll teach more about locking and concurrency, and read more papers.

End of Quiz II