EE516 : Project 3

Gaurav Kalra

(Student ID: 2016 45 93)

gvkalra@kaist.ac.kr

```c
static void
eat_if_possible(int id)
{
    if (phil_state[id] == PHIL_STATE_HUNGRY /* 'id' is hungry */
        && phil_state[PHIL_LEFT(id)] != PHIL_STATE_EATING /* left is not eat
        && phil_state[PHIL_RIGHT(id)] != PHIL_STATE_EATING) /* right is not
    {
        phil_state[id] = PHIL_STATE_EATING;
        /* Notice that PHIL_LEFT(id) has the same value as left side fork
         * so we can re-use the same macro */
        info("Philosopher [%d] : FORK_UP (%d, %d)", id, PHIL_LEFT(id), id);

        /* signal eating = True for philosopher 'id' */
        sem_post(&phil_sema[id]);
    }
}

static void
take_forks(int id)
{
    sem_wait(&mutex);
    phil_state[id] = PHIL_STATE_HUNGRY;
    eat_if_possible(id);
    sem_post(&mutex);

    /* wait until philosopher 'id' is allowed to eat
     * who is going to tell if 'id' is allowed to eat now?
     *     it's neibouring philosophers in put_forks()
     */
    sem_wait(&phil_sema[id]);
}
```

Figure 1: Taking forks

Key Point:

When trying to take forks, a philosopher will “eat_if_possible”. Otherwise, it will wait to be woken up by it’s neighbouring philosophers.

```c
static void
put_forks(int id)
{
    sem_wait(&mutex);
    phil_state[id] = PHIL_STATE_THINKING;
    info("Philosopher [%d] : FORK_DOWN (%d, %d)", id, PHIL_LEFT(id), id);

    /* see if neighbouring philosophers can eat
     * this means -> neighbour should be HUNGRY
     * and waiting on it's semaphore (phil_sema)
     */
    eat_if_possible(PHIL_LEFT(id)); /* left can eat? */
    eat_if_possible(PHIL_RIGHT(id)); /* right can eat? */
    sem_post(&mutex);
}
```

Figure 2: Releasing forks

Key Point:

After finishing eating, a philosopher will check if it's neighbouring (left & right) philosophers can "eat_if_possible". If so, they will be woken up to start eating.

```
=24338== Process terminating with default action of signal 2 (SIGINT)
=24338==    at 0x4E489DD: pthread_join (pthread_join.c:90)
=24338==    by 0x4C31DE5: pthread_join_WRK (hg_intercepts.c:553)
=24338==    by 0x400BBC: main (dining_philosopher.c:102)
=24338==
=24338== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2296 from 110)
-24338--
-24338-- used_suppression:    993 helgrind-glibc2X-004 /usr/lib/valgrind/default.supp:931
-24338-- used_suppression:   1299 helgrind-glibc-io-xsputn-mempcpy /usr/lib/valgrind/default.supp:937
-24338-- used_suppression:      4 helgrind-glibc2X-101 /usr/lib/valgrind/default.supp:980
=24338==
=24338== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2296 from 110)
illed
```

Figure 3: Verification using helgrind

```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR03/task01/philosopher (master) $ ./dining_philosopher
<philosopher:167> Running Thread for philosopher: [0]
Philosopher [0] : Thinking
<philosopher:167> Running Thread for philosopher: [1]
Philosopher [1] : Thinking
<philosopher:167> Running Thread for philosopher: [2]
Philosopher [2] : Thinking
<philosopher:167> Running Thread for philosopher: [3]
Philosopher [3] : Thinking
<philosopher:167> Running Thread for philosopher: [4]
Philosopher [4] : Thinking
Philosopher [2] : FORK_UP (1, 2)
Philosopher [2] : Eating
Philosopher [0] : FORK_UP (4, 0)
Philosopher [0] : Eating
Philosopher [2] : FORK_DOWN (1, 2)
Philosopher [3] : FORK_UP (2, 3)
Philosopher [2] : Thinking
Philosopher [3] : Eating
Philosopher [0] : FORK_DOWN (4, 0)
Philosopher [1] : FORK_UP (0, 1)
Philosopher [0] : Thinking
Philosopher [1] : Eating
```

Figure 4: Sample Output

Task 1 (P2): Solve "training monkey" problem using Linux semaphore

```c
enum {
    MONKEY_STATE_BORN = 0, /* monkey is just born */
    MONKEY_STATE_READY_ROOM_TRAINING = 1, /* ready to be trained in room */
    MONKEY_STATE_ROOM_TRAINING = 2, /* room training is on-going */
    MONKEY_STATE_THINKING = 3, /* thinking */
    MONKEY_STATE_ROOM_TRAINED = 4, /* room training finished */
};

enum {
    BALL_COLOR_INVALID = -1,
    BALL_COLOR_RED = 0,
    BALL_COLOR_GREEN = 1,
    BALL_COLOR_BLUE = 2,
    BALL_COLOR_YELLOW = 3,
    BALL_COLOR_MAX
};

static const char *color_string[] = {
    [BALL_COLOR_RED] = "RED",
    [BALL_COLOR_GREEN] = "GREEN",
    [BALL_COLOR_BLUE] = "BLUE",
    [BALL_COLOR_YELLOW] = "YELLOW",
};

struct {
    int state; /* MONKEY_STATE_* enumeration */
    int color[2]; /* colors (BALL_COLOR_*) of ball interested in */
} monkey_data[MONKEY_TOTAL] = {
    {MONKEY_STATE_BORN, {BALL_COLOR_INVALID, BALL_COLOR_INVALID}},
};
```

Figure 1: Data Structure of a Monkey (**struct monkey_data**)

```c
sem_t room; /* for access to room */
sem_t mutex; /* for critical section */
sem_t monkey_sema[MONKEY_TOTAL]; /* for each monkey */

sem_t bowl; /* banana bowl */
sem_t trainer; /* trainer */
```

```c
/* initialize 'monkey_sema' */
for (i = 0; i < MONKEY_TOTAL; i++) {
    /* initial value of 0 indicates that
     * no monkey is training */
    res = sem_init(&monkey_sema[i], 0, 0);
```

```c
/* initialize 'room'
 * initial value 4 means 4 monkeys can enter
 * the room
 */
res = sem_init(&room, 0, 4);
```

```c
/* initialize 'trainer' (there is only 1 trainer) */
res = sem_init(&trainer, 0, 1);
```

```c
/* initialize 'mutex' */
res = sem_init(&mutex, 0, 1);
```

```c
/* initialize 'bowl'
 * initial value 2 means 2 bowls are available
 * outside the room
 */
res = sem_init(&bowl, 0, 2);
```

Figure 2: Semaphores for room, data, monkey, bowl & trainer

```c
static void
release_balls(int id)
{
    int i;

    sem_wait(&mutex);
    monkey_data[id].state = MONKEY_

    info("Monkey %d (%s, %s): balls
        color_string[monkey_data[id
        color_string[monkey_data[id

    /* see if other monkeys in room
    for (i = 0; i < MONKEY_TOTAL; i
        if (monkey_data[i].state ==
            __train_if_you_can(i);
        }
```

Key Point:

After finishing training (**release_balls**), a monkey will check if other monkeys inside the room can train themselves (**__train_if_you_can**). If so, they will be woken up to start training.

```
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR03/task01/monkey (master) $ ./training_monkey
Monkey 4 (BLUE, GREEN): entered
Monkey 0 (BLUE, YELLOW): entered
Monkey 2 (RED, BLUE): entered
Monkey 3 (RED, BLUE): entered
Monkey 4 (BLUE, GREEN): takes the BLUE ball
Monkey 4 (BLUE, GREEN): thinking
Monkey 4 (BLUE, GREEN): takes the GREEN ball
Monkey 4 (BLUE, GREEN): thinking
Monkey 4 (BLUE, GREEN): balls released
Monkey 0 (BLUE, YELLOW): takes the BLUE ball
Monkey 0 (BLUE, YELLOW): thinking
Monkey 0 (BLUE, YELLOW): takes the YELLOW ball
Monkey 4 (BLUE, GREEN): left
Monkey 1 (GREEN, YELLOW): entered
Trainer: puts bananas
Trainer: goes to sleep
Monkey 4 (BLUE, GREEN): eat bananas
Monkey 0 (BLUE, YELLOW): thinking
Monkey 0 (BLUE, YELLOW): balls released
Monkey 1 (GREEN, YELLOW): takes the GREEN ball
Monkey 1 (GREEN, YELLOW): thinking
Monkey 1 (GREEN, YELLOW): takes the YELLOW ball
Monkey 2 (RED, BLUE): takes the RED ball
Monkey 2 (RED, BLUE): thinking
Monkey 1 (GREEN, YELLOW): thinking
Monkey 2 (RED, BLUE): takes the BLUE ball
Monkey 0 (BLUE, YELLOW): left
```

Figure 3: Sample Output

```
Trainer: goes to sleep
Monkey 29 (RED, BLUE): eat bananas
--16886-- REDIR: 0x4e4ef20 (libpthread.so.0:sem_destroy@@GLIBC_2.2.5) redirected to 0x4c366b0 (sem_destroy@*)
--16886-- REDIR: 0x4e49d90 (libpthread.so.0:pthread_mutex_lock) redirected to 0x4c360e0 (pthread_mutex_lock)
--16886-- REDIR: 0x4e4b510 (libpthread.so.0:pthread_mutex_unlock) redirected to 0x4c36110 (pthread_mutex_unlock)
==16886==
==16886== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 10128 from 235)
--16886--
--16886-- used_suppression:   4252 helgrind-glibc2X-004 /usr/lib/valgrind/default.supp:931
--16886-- used_suppression:   5847 helgrind-glibc-io-xsputn-mempcpy /usr/lib/valgrind/default.supp:937
--16886-- used_suppression:     29 helgrind-glibc2X-101 /usr/lib/valgrind/default.supp:980
==16886==
==16886== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 10128 from 235)
gvkalra@gvkalra-desktop ~/Desktop/EE516/PR03/task01/monkey (master) $
```

Figure 4: Verification using helgrind

Task 2: Make your own system call

User gives two integer values, system handler prints information + student ID + returns sum of integers, user prints the result

```
From 827324e5504bd239e9e464f86a9fe48f94c6d20f Mon Sep 17 00:00:00 2001
From: Gaurav Kalra <gvkalra@kaist.ac.kr>
Date: Sat, 5 Nov 2016 19:48:36 +0900
Subject: [PATCH] PR03-task02 : Make your own system call

---
 arch/x86/syscalls/syscall_64.tbl |  3 +++
 include/linux/syscalls.h         |  1 +
 kernel/Makefile                  |  3 ++-
 kernel/mysyscall.c               | 11 +++++++++++
 4 files changed, 17 insertions(+), 1 deletion(-)
 create mode 100644 kernel/mysyscall.c
```

Figure 1: Patch Summary

```
diff --git a/arch/x86/syscalls/syscall_64.tbl b/arch/x86/syscalls/syscall_64.tbl
index 281150b..3036770 100644
--- a/arch/x86/syscalls/syscall_64.tbl
+++ b/arch/x86/syscalls/syscall_64.tbl
@@ -329,6 +329,9 @@
 320    common  kexec_file_load         sys_kexec_file_load
 321    common  bpf             sys_bpf

+# added by gvkalra
+322    common  mysyscall       sys_mysyscall
+
 #
 # x32-specific system call numbers start at 512 to avoid cache impact
 # for native 64-bit operation.
```

Figure 2: syscall_64.tbl

```
diff --git a/include/linux/syscalls.h b/include/linux/syscalls.h
index bda9b81..b84e05a 100644
--- a/include/linux/syscalls.h
+++ b/include/linux/syscalls.h
@@ -877,4 +877,5 @@ asmlinkage long sys_seccomp(unsigned int op, unsigned int flags,
 asmlinkage long sys_getrandom(char __user *buf, size_t count,
                                unsigned int flags);
 asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);
+asmlinkage long sys_mysyscall(int a, int b);
 #endif
```

Figure 3: syscalls.h

```
diff --git a/kernel/Makefile b/kernel/Makefile
index 17ea6d4..fe3ec0d 100644
--- a/kernel/Makefile
+++ b/kernel/Makefile
@@ -9,7 +9,8 @@ obj-y     = fork.o exec_domain.o panic.o \
            extable.o params.o \
            kthread.o sys_ni.o nsproxy.o \
            notifier.o ksysfs.o cred.o reboot.o \
-           async.o range.o groups.o smpboot.o
+           async.o range.o groups.o smpboot.o \
+           mysyscall.o

 ifdef CONFIG_FUNCTION_TRACER
 # Do not trace debug files and internal ftrace files
```

Figure 4: Makefile

```
diff --git a/kernel/mysyscall.c b/kernel/mysyscall.c
new file mode 100644
index 0000000..00cc379
--- /dev/null
+++ b/kernel/mysyscall.c
@@ -0,0 +1,11 @@
+#include <linux/unistd.h>
+#include <linux/errno.h>
+#include <linux/kernel.h>
+#include <linux/sched.h>
+
+asmlinkage long sys_mysyscall(int a, int b)
+{
+    printk("Student ID: 20164593\n");
+    printk("mysyscall: a=%d, b=%d\n", a, b);
+    return a + b;
+}
\ No newline at end of file
```

Figure 5: mysyscall.c

```
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>

#define __NR_mycall 322

int main(int argc, const char *argv[])
{
    int n;

    n = syscall(__NR_mycall, 5, 15);
    printf("mycall return value : %d\n", n);

    return 0;
}
```

Figure 6: syscall_test.c (user program)

```
gvkalra@ubuntu:~/Desktop/EE516/PR03/task02$ ./syscall_test
mycall return value : 20
```

Figure 7: User Program execution

```
[    8.490434] input: VMware VMware Virtual USB Mouse as /devices/pci0000:00/0000:00:11.0/0000:02:0
[    8.491606] hid-generic 0003:0E0F:0003.0001: input,hidraw0: USB HID v1.10 Mouse [VMware VMware V
[    8.522505] audit_printk_skb: 39 callbacks suppressed
[    8.522505] audit: type=1400 audit(1478593888.696:24): apparmor="STATUS" operation="profile_load
[    8.522717] audit: type=1400 audit(1478593888.696:25): apparmor="STATUS" operation="profile_load
[    8.522874] audit: type=1400 audit(1478593888.696:26): apparmor="STATUS" operation="profile_load
[    8.522963] audit: type=1400 audit(1478593888.696:27): apparmor="STATUS" operation="profile_load
[    8.523581] audit: type=1400 audit(1478593888.696:28): apparmor="STATUS" operation="profile_load
[   10.047545] audit: type=1400 audit(1478593890.220:29): apparmor="STATUS" operation="profile_load
[   10.047755] audit: type=1400 audit(1478593890.220:30): apparmor="STATUS" operation="profile_load
[   10.048063] audit: type=1400 audit(1478593890.220:31): apparmor="STATUS" operation="profile_load
[   10.048174] audit: type=1400 audit(1478593890.220:32): apparmor="STATUS" operation="profile_load
[   10.048406] audit: type=1400 audit(1478593890.220:33): apparmor="STATUS" operation="profile_load
[   10.307456] floppy0: no floppy controllers found
[   10.307477] work still pending
[   10.507721] init: plymouth-upstart-bridge main process ended, respawning
[   76.749802] Student ID: 20164593
[   76.749808] mysyscall: a=5, b=15
```

Figure 8: Kernel Logs

Task 3: Make your own semaphores (refer problem statement for specifications)

```
From 7ea10138a3991734531c1bf99195797383c14cc5 Mon Sep 17 00:00:00 2001
From: Gaurav Kalra <gvkalra@kaist.ac.kr>
Date: Tue, 8 Nov 2016 21:35:03 +0900
Subject: [PATCH] Custom semaphore

---
 arch/x86/syscalls/syscall_64.tbl |   7 +
 include/linux/syscalls.h         |   6 +
 kernel/Makefile                  |   3 +-
 kernel/mysemaphore.c             | 380 ++++++++++++++++++++++++++++++++++++
 4 files changed, 395 insertions(+), 1 deletion(-)
 create mode 100644 kernel/mysemaphore.c
```

Figure 1: Patch Summary

```
diff --git a/arch/x86/syscalls/syscall_64.tbl b/arch/x86/syscalls/syscall_64.tbl
index 281150b..e867861 100644
--- a/arch/x86/syscalls/syscall_64.tbl
+++ b/arch/x86/syscalls/syscall_64.tbl
@@ -329,6 +329,13 @@
 320	common	kexec_file_load		sys_kexec_file_load
 321	common	bpf			sys_bpf

+# added by gvkalra
+322	common	mysema_init		sys_mysema_init
+323	common	mysema_down		sys_mysema_down
+324	common	mysema_down_userprio	sys_mysema_down_userprio
+325	common	mysema_up		sys_mysema_up
+326	common	mysema_release		sys_mysema_release
+
 #
 # x32-specific system call numbers start at 512 to avoid cache impact
 # for native 64-bit operation.
```

Figure 2: syscall_64.tbl

```
diff --git a/kernel/Makefile b/kernel/Makefile
index 17ea6d4..62c1665 100644
--- a/kernel/Makefile
+++ b/kernel/Makefile
@@ -9,7 +9,8 @@ obj-y     = fork.o exec_domain.o panic.o \
	    extable.o params.o \
	    kthread.o sys_ni.o nsproxy.o \
	    notifier.o ksysfs.o cred.o reboot.o \
-	    async.o range.o groups.o smpboot.o
+	    async.o range.o groups.o smpboot.o \
+	    mysemaphore.o

 ifdef CONFIG_FUNCTION_TRACER
 # Do not trace debug files and internal ftrace files
```

Figure 3: Makefile

```
diff --git a/include/linux/syscalls.h b/include/linux/syscalls.h
index bda9b81..d347dad 100644
--- a/include/linux/syscalls.h
+++ b/include/linux/syscalls.h
@@ -877,4 +877,10 @@ asmlinkage long sys_seccomp(unsigned int op, unsigned int flags,
 asmlinkage long sys_getrandom(char __user *buf, size_t count,
			      unsigned int flags);
 asmlinkage long sys_bpf(int cmd, union bpf_attr *attr, unsigned int size);
+
+asmlinkage int sys_mysema_init(int sema_id, int start_value, int mode);
+asmlinkage int sys_mysema_down(int sema_id);
+asmlinkage int sys_mysema_down_userprio(int sema_id, int priority);
+asmlinkage int sys_mysema_up(int sema_id);
+asmlinkage int sys_mysema_release(int sema_id);
 #endif
```

Figure 4: syscalls.h

```
gcc -o semaphore_test semaphore_test.c -Wall -Werror -g -lpthread
gvkalra@gvkalra-vbox:~/Desktop/EE516/PR03/task03$ ./semaphore_test
test_init_release() : PASSED
test_fifo() : PASSED
test_user_prio() : PASSED
test_os_prio() : PASSED
```

Figure 5: ./semaphore_test (user program)

```
[ 2070.681317] Semaphore (0) initialized
[ 2070.681332] Already active
[ 2070.681343] Semaphore is not active
[ 2070.681350] Semaphore (1) initialized
[ 2070.681357] Already active
[ 2070.681363] Semaphore is not active
[ 2070.681370] Semaphore (2) initialized
[ 2070.681376] Already active
[ 2070.681383] Semaphore is not active
[ 2070.681389] Semaphore (3) initialized
[ 2070.681396] Already active
[ 2070.681403] Semaphore is not active
[ 2070.681410] Semaphore (4) initialized
[ 2070.681481] Already active
[ 2070.681489] Semaphore is not active
[ 2070.681495] Semaphore (5) initialized
[ 2070.681501] Already active
[ 2070.681508] Semaphore is not active
[ 2070.681514] Semaphore (6) initialized
[ 2070.681520] Already active
```

Figure 6: dmesg logs

```
gvkalra@gvkalra-vbox:~/Desktop/EE516/PR03/task03$ uname -a
Linux gvkalra-vbox 3.18.21+ #4 SMP Sat Nov 12 22:20:55 KST 2016 x86_64 x86_64 x86_64 GNU/Linux
```

Figure 7: Kernel build information

```
+struct mysemaphore {
+	raw_spinlock_t lock;
+	unsigned int value;
+	unsigned int state; /* 0: i
+	unsigned int mode; /* 0: FI
+	struct list_head wait_list;
+};
+
+struct mysemaphore_waiter {
+	struct list_head list;
+	struct task_struct *task;
+	unsigned int priority;
+	bool up;
+};
```

Key Point:

Each semaphore maintains a list of “tasks” contesting claim on a semaphore. “mysemaphore_waiter” represents an entry of such a task. User priority is saved in waiter list.

Each semaphore is protected by a spin lock.

```
+static inline int
+_down_common(struct mysemaphore *sem, long state,
+    long timeout, unsigned int priority)
+{
+    struct task_struct *task = current;
+    struct mysemaphore_waiter waiter;
+
+    list_add_tail(&waiter.list, &sem->wait_list);
+    waiter.task = task;
+    waiter.priority = priority;
+    waiter.up = false;
+
+    for (;;) {
+        if (signal_pending_state(state, task))
+            goto interrupted;
+        if (unlikely(timeout <= 0))
+            goto timed_out;
+        __set_task_state(task, state);
+        raw_spin_unlock_irq(&sem->lock);
+        timeout = schedule_timeout(timeout);
+        raw_spin_lock_irq(&sem->lock);
+        if (waiter.up)
+            return 0;
+    }
+
+timed_out:
+    list_del(&waiter.list);
+    return -1;
```

Key Point:
A request for a semaphore is added to wait_list. Using scheduler APIs, the task is put to sleep until “waiter.up” becomes “true”. In my implementation, timeout is set to MAX_SCHEDULE_TIMEOUT and interruptible is set to TASK_UNINTERRUPTIBLE.

```
+    /* FIFO */
+    if (mode == 0) {
+        /* wakeup the first task in queue */
+        waiter = list_first_entry(&sem->wait_list,
+            struct mysemaphore_waiter, list);
+    }
+    /* OS priority */
+    else if (mode == 1) {
+        waiter = _get_lowest_prio(sem);
+    }
+    /* User priority */
+    else if (mode == 2) {
+        waiter = _get_highest_user_prio(sem);
+    } else {
+        printk(KERN_ERR "UNKNOWN ERROR!\n");
+        return;
+    }
+
+    list_del(&waiter->list);
+    waiter->up = true;
+    wake_up_process(waiter->task);
```

Key Point:
Waiting processes are woken up (by setting “waiter.up” to “true”) according to the “mode” setting. **_get_lowest_prio()** will return the waiting process with lowest “prio” in task_struct.

_get_highest_user_prio() will return the waiting process with highest “priority” in waiter structure.

Q1. For "Dining philosopher" problem, how can you prevent starvation?
Starvation may be prevented by giving preferential treatment to the most "starved" philosopher. And a disadvantage to the philosopher that has just eaten. In other words, philosophers may not be allowed to eat twice in a row without letting others use the forks in between.

Q2. For "Training monkey" problem, if two male monkeys and two female monkeys can stay in the room, how should your solution be modified?
In my current solution, all monkeys compete for entering the room equally. In other words, all monkeys are competing for 'room' semaphore, which is initialized to 4 (since 4 monkeys can be present in the room at the same time). If however, we need to distinguish between male & female monkeys, we can model this by assuming two different semaphores (one for each gender). Male monkeys outside the room will compete for "male_semaphore" & female monkeys outside the room will compete for "female_semaphore". In essence, it will be similar to having two different queues (male, female) for entering the training room. Once inside the room, monkeys compete for same balls irrespective of gender.

Q3. For "Training monkey" problem, if monkeys can enter the room in the ascending order of their IDs, how should your solution be modified?
In my current solution, all monkeys compete for entering the room equally. In other words, all monkeys are competing for 'room' semaphore, which is initialized to 4 (since 4 monkeys can be present in the room at the same time). If however, monkeys can only enter the room in the ascending order of their IDs (and we assume all monkeys are ready outside the room), I will not model monkeys as threads. Since there is a strict order for resource usage (room in this case) already defined, it makes more sense to model "monkey cages inside the room" as threads. In other words, since there are 4 monkeys allowed to be in room at any given time, model these "4 spots" as threads. These 4 threads will read & write to a common data structure (database of monkeys). Similarly, if we assume a total order for using 'bowls' as well, we can model bowls as threads which read & write to a shared data structure of monkeys.

Q4. There are POSIX semaphore and non-POSIX semaphore (ex. System V semaphore). What is the difference? Pros and cons?
POSIX is a standard defining APIs, command line shells & utility interfaces for software compatibility with variants of Unix and other operating systems. As such, POSIX is not the only standard in existence. System V (often known as SysV) is also one of the standards. Although both POSIX & non-POSIX standard provide almost the same "tools" (e.g. semaphores, shared memory and message queues), they offer different interfaces to those tools.

As an example, SysV provides the following abstractions (system calls) for semaphore:
semget() - to create a new semaphore set, or access an existing set
semop() - to perform specified operations on selected semaphores
semctl() - to perform control operations on a semaphore set
etc ...

Whereas, POSIX provides the following abstractions (system calls) for semaphore:
sem_init() - to initialize a semaphore
sem_wait() - to decrement & wait on a semaphore
sem_post() - to increment & wakeup waiting processes on a semaphore
etc ...

In essence, the difference between POSIX & non-POSIX semaphores is of APIs & implementation in kernel for the same concept.

However, there are a number of subtle pros & cons:

1. In System V you can control how much the semaphore count can be increased or decreased; whereas in POSIX, the semaphore count is increased and decreased by 1.
2. POSIX semaphores do not allow manipulation of semaphore permissions, whereas System V semaphores allow you to change the permissions of semaphores to a subset of the original permission.
3. Initialization and creation of semaphores is atomic (from the user's perspective) in POSIX semaphores.
4. From a usage perspective, System V semaphores are clumsy, while POSIX semaphores are straight-forward
5. The scalability of POSIX semaphores (using unnamed semaphores) is much higher than System V semaphores. In a user/client scenario, where each user creates her own instances of a server, it would be better to use POSIX semaphores.
6. System V semaphores, when creating a semaphore object, creates an array of semaphores whereas POSIX semaphores create just one. Because of this feature, semaphore creation (memory footprint-wise) is costlier in System V semaphores when compared to POSIX semaphores.
7. It has been said that POSIX semaphore performance is better than System V-based semaphores.
8. POSIX semaphores provide a mechanism for process-wide semaphores rather than system-wide semaphores. So, if a developer forgets to close the semaphore, on process exit the semaphore is cleaned up. In simple terms, POSIX semaphores provide a mechanism for non-persistent semaphores.

References:

http://www.tldp.org/LDP/lpg/node46.html
https://linux.die.net/include/semaphore.h
http://www.linuxdevcenter.com/pub/a/linux/2007/05/24/semaphores-in-linux.html?page=4

Q5. Classify semaphores in task 1 and task 3 into POSIX and non-POSIX semaphore. What is the reason?
Semaphores in task 1 are POSIX semaphores (since they provide APIs & implementation as stated in POSIX standard). On the other hand, task 3 semaphores are _not_ POSIX semaphores. This is because the APIs & implementation is not according to the POSIX standard.

References:

http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/semaphore.h.html

Q6. An user-level process may access to kernel or hardware directly. What are pros and cons of using system call?
Although it is possible to access hardware directly from user-level process (e.g. by using root account), it is not advisable. This is because there can be situations, where more than 1 user-level process require access to hardware. In this case, we need to synchronize between different processes for hardware access. If we implement hardware access in kernel, it is easier to synchronize between multiple processes. Also, accessing hardware using system call facilitates for easy code reuse.

Cons:

1. You need a syscall number, which needs to be officially assigned to you during a developmental kernel series.
2. System calls are not easily used from scripts and cannot be accessed directly from the filesystem.
3. For simple exchanges of information, a system call is overkill.

For many interfaces, system calls are the correct answer. Linux, however, has tried to avoid simply adding a system call to support each new abstraction that comes along. The result has been an incredibly clean system call layer with very few regrets or deprecations (interfaces no longer used or supported). The slow rate of addition of new system calls is a sign that Linux is a relatively stable and feature-complete operating system.

References:

http://www.makelinux.net/books/lkd2/ch05lev1sec5

Q7. To make a new system call, we compiled entire kernel. What happen if we try to do it with adding a module?
It is not possible because system call table (sys_call_table) is a static size array. And its size is determined at compile time by the number of registered syscalls. This means there is no space for another one.

There are a few hacks which enable us to add a new system call with a module:
1. Change your kernel to export sys_call_table symbol to modules.
2. Find syscall table dynamically - Iterate over kernel memory, comparing each word with a pointer to known system call function.

However, none of these are recommended in practice and should only be used as only a fun way to play with kernel.

References:

http://unix.stackexchange.com/a/48208

Q8. Your own semaphore has several modes. What is pros and cons of each mode? In what situations each mode can take a benefit?
My own semaphore has 3 modes:

1. FIFO

Pros:
- i. Fairness - The process/thread requesting the resource will get it in same order as requested.
- ii. Easy to implement - We don't need to maintain any additional data or do extra processing when waking up the process.

Cons:
- i. No way to specify priority - Another process may be in urgent need of a shared resource.

FIFO mode may be beneficial in general non-real time systems.

2. User priority

Pros:
- i. Provides flexibility to user for specifying priority of process to acquire shared resource

Cons:
- i. A spurious process may always request for HIGH priority making other processes starve
- ii. It requires maintaining additional information of priority with every request for a shared resource

User priority mode is beneficial when all user-space processes are trusted to specify correct priority.

3. OS priority

Pros:
- i. It gives resource assignment priority to HIGH priority process

Cons:
- i. It requires processing priority of all waiting processes, which is an additional overhead.
- ii. Priority of a process may not reflect urgency of resource requirement

OS priority mode is beneficial in systems where priority of a process can be equated to urgency of resource requirement