

CS 3204 Sample Midterm

Answers are shown in this style. This exam was given in Spring 2007.

1. Synchronization (16 pts)

- a) (8 pts) A well in a desert village is only big enough to allow 5 villagers at a time to draw water from it. If more than 5 villagers tried to get water simultaneously, a brawl would occur and everyone would suffer.

Having recently learned that Computers can help solve problems, the village elders have hired you to help them. You decide to model the villager's problem using a computer program. Assuming that each villager is represented by a thread, the following code would be run by each thread.

```
villager_thread()
{
    while(1)
    {
        waitInLine();
        // get water
        returnHome();
        // use water
    }
}
```

Using Pintos-style semaphores write the C code for the `waitInLine()` and `returnHome()` functions to avoid a fight at the well.

(2 pts) // insert variable declaration(s)
// + initialization here

```
struct semaphore accessToWell;
sema_init(&accessToWell, 5);
```

(3 pts)

```
void waitInLine() {
    sema_down(&accessToWell);
}
```

(3 pts)

```
void returnHome() {
```

```

        sema_up(&accessToWell);
    }

```

- b) (8 pts) Would it be possible to implement a solution to this problem *exclusively* with Pintos-style locks? Your solution should be race-condition free and avoid busy waiting. If so, sketch the solution. If not, say why not.

A solution would require direct access to thread_block/unblock to avoid busy waiting, enforce precedence constraints, and handle any lost wakeups. This leads to a very complicated design. For these reasons, it is not advisable to use locks for this problem – instead, semaphores or monitors (using condition variables) should be used.

Correct solutions would likely reimplement the functionality these higher-level synchronization constructs provide, for instance, see below for an implementation that would mimic a monitor and the wait/broadcast paradigm.

```

// initialization
Number of available well slots = 5;
lock_init(accessToWell);

```

<pre> void waitInLine() { lock_acquire(accessToWell); while(no avail well slots) { waiters.add(current); lock_release(well); thread_block(current); lock_acquire(accessToWell); } Number of well slots--; waiters.remove(current); lock_release(accessToWell); } </pre>	<pre> void returnHome() { lock_acquire(accessToWell); Number of well slots++; if (num waiters > 0) { call thread_unblock on all waiters; } lock_release(accessToWell); } </pre>
---	--

Accepted answers included ‘yes’ – if the implementation was sketched correctly, or ‘no’, if the argument was made that locks by themselves do not allow for the expression of precedence constraints and that they do not support limiting the number of threads accessing a resource simultaneously, unless the limit is 1. Solutions that used one lock to allow only one person to use the well at a time did not receive full credit, as this is grossly inefficient.

A common mistake was to propose to use 5 locks, however, this would mean that a person could be blocked waiting for one slot even though other slots have become free. Some have tried to fix this using lock_try_acquire() – but in this case, the solution suffers from busy waiting.

2. Named Pipes (40 pts)

Suppose you wanted to add support for first-in-first-out (fifo) special files to Pintos. In the POSIX world, such special files are also referred to as *named pipes*. They provide a way for two processes to communicate reliably: one process (the ‘reader’) opens the pipe in read-only mode, a second process (the ‘writer’) opens the pipe in write-only mode. The kernel passes all data that is written by the writer via the `write()` system call on to the reader. This is done using a *bounded buffer* that the kernel allocates internally – no data is ever written to the actual file system. To support named pipes, they have to be first created and then opened by either end.

Here’s an example of how they can be used, where `#include` files and error checking have been omitted for brevity.

```
// namedpipeexample.c
int
main(int ac, char *av[])
{
    mkfifo("apipe");
    exec("./reader apipe");
    exec("./writer apipe namedpipeexample.c");
    // not shown: namedpipeexample waits for its children
}

// reader.c
int
main(int ac, char *av[])
{
    int readend = fifo_open_read(av[1]);

    int bytes_read;
    char buf[80];
    while ((bytes_read = read(readend, buf, sizeof buf)) > 0)
        write (1, buf, bytes_read);
}

// writer.c
int
main(int ac, char *av[])
{
    int writeend = fifo_open_write(av[1]);
    int fd = open(av[2], O_RDONLY);

    int bytes_to_write;
    char buf[80];
    while ((bytes_to_write = read(fd, buf, sizeof buf)) > 0)
        write (writeend, buf, bytes_to_write);
}
```

When “namedpipeexample” is run, the resulting output should be a dump of its source code, contained in `namedpipeexample.c`: the “writer” process will read the `.c` file, write it to the named pipe, from where the “reader” process will read it and write it to the console via its `stdout` file descriptor.

The prototypes for the system calls you must implement are shown here:

```
// creates a new named pipe, returns true on success
bool mkfifo(const char *name);

// opens a named pipe for reading
// returns a fd if successful, -1 otherwise
int fifo_open_read(const char *name);

// opens a named pipe for writing
// returns a fd if successful, -1 otherwise
int fifo_open_write(const char *name);
```

- a) (4 pts) To implement these three new system calls, explain what support you have to provide to the standard C library with which Pintos user programs are linked such that Pintos processes will be able to use these new calls. Name 1 item.

Pintos’s standard C library would have to be extended with stubs that would allow user processes to make those system calls. In addition, new system call numbers will have to be defined and prototypes be added to the header files such that user programs can link against these stubs such that system calls appear as ordinary function calls to user programs. These changes concern `lib/user/syscall.c`

- b) (6 pts) What changes would you have to make to your system call dispatch framework on the kernel side to handle these additional system calls? Specifically, address how you ensure that a user process cannot trick your kernel into accessing another process’s data!

*You’ll need to add new entries to your system call dispatch table (in `userprog/syscall.c`), or, if you are using a `switch()` statement, new arms to this statement. You will have to check the validity of the system call number, and you will have to check the validity of the zero-terminated string to which the user process provides a pointer (all three calls take a “const char *”). This checking involves making sure that all addresses starting from `name` until the first address that contains a zero byte are valid user space addresses for this process.*

- c) (8 pts) Explain how you would implement `mkfifo()`, `fifo_open_read()`, and `fifo_open_write()`. First, describe the data structure(s) you would need to add to your kernel. Suppose you represented each created fifo object in a data structure `struct fifo`. Sketch what members this structure would contain:

You'll have to allocate a buffer and some synchronization objects. 1 condition variable + 1 lock would work, as would 1 lock + 2 semaphore, or 3 semaphores. [For the bounded buffer problem, a 2 semaphore solution is also possible, but needs to be justified.]

```
struct fifo {
    /* bounded buffer to hold data written but not read */
    char buffer[4096];
    /* indexes reflecting the current amount of data */
    int head, tail;
    /* lock to protect other members in this struct */
    struct lock fifolock;
    /* condition variable to signal change in buffer state */
    struct condition pipestatechange;
}
```

For the remaining parts, you may assume the existence of a dictionary for fifo objects that provides these functions:

```
// add a new fifo object, return true if successful
bool fifo_add(const char *name, struct fifo *fifo);

// retrieve a previously added fifo by name,
// or NULL if it does not exist
struct fifo *fifo_get(const char *name);
```

d) (4 pts) Sketch your implementation of `mkfifo()` – use C or pseudocode.

```
bool mkfifo(const char *name) {
    char *kname = copyin_string(name);
    if (!kname)
        return false;
    struct fifo * f = malloc(sizeof (*f));
    f->head = f->tail = 0;
    lock_init(&f->fifolock);
    cond_init(&f->pipestatechange);
    bool success = fifo_add(kname, f);
    free (kname);
    return success;
}
```

e) (6 pts) Sketch your implementation `fifo_open_read()` and `fifo_open_write()` – use C or pseudocode. In particular, address how the representation of your file descriptors would have to change. (You may use functions defined in your project submission if their names are self-explanatory.)

Your file descriptor implementation will now need to include some identifier/flag for each file descriptor that says whether this file descriptor refers to a named pipe or a regular file. You'll also have to distinguish between a read and write end

of the pipe. Let's assume you've added a parameter "flag" to your allocate_fd function that specifies this.

```
int fifo_open(const char *name, int flag) {
    int fd = -1;
    char *kname = copyin_string(name);
    if (kname) {
        struct fifo * f = fifo_get(kname);
        if (f) {
            fd = allocate_fd(f, flag);
        }
        free(kname);
    }
    return fd;
}

int fifo_open_read(const char *name) {
    return fifo_open(name, FIFO_READ);
}

int fifo_open_write(const char *name) {
    return fifo_open(name, FIFO_WRITE);
}
```

- f) (6 pts) You will need to change your `write()` system call. Sketch how. Make sure that there are no race conditions. Make sure that your implementation does not limit the amount of data a process can pass to `write()`, and ensure that no data is lost.

Your write() will need to identify when a write is issued for the write end of a pipe, and if so, you'll have to implement a straight bounded-buffer.

```
int write(int fd, const void *buffer, size_t len) {
    struct fifo * pipe = fd_get_pipe_write_end(fd);
    if (pipe)
        return pipe_write(pipe, buffer, len);
    // else handle remaining cases
}

int pipe_write(struct fifo *pipe, const void *buffer, size_t len) {
    int total_written = 0;
    lock_acquire(&pipe->lock);
    while (len > 0) {
        while (fifo_isfull(pipe))
            cond_wait(&pipe->pipestatechange, &pipe->lock);
        int bytes_copied = fifo_copy_into(pipe, buffer, len);
        buffer += bytes_copied;
        len -= bytes_copied;
        total_written += bytes_copied;
        cond_signal(&pipe->pipestatechange, &pipe->lock);
    }
    lock_release(&pipe->lock);
    return total_written;
}
```

- g) 6 pts) Sketch how your `read()` system call implementation will have to change. Make sure that proper synchronization is used.

```
int read(int fd, void *buffer, size_t len) {
    struct fifo * pipe = fd_get_pipe_read_end(fd);
    if (pipe)
        return pipe_read(pipe, buffer, len);
    // else handle remaining cases
}

int pipe_read(struct fifo *pipe, void *buffer, size_t len) {
    lock_acquire(&pipe->lock);
    while (fifo_isempty(pipe))
        cond_wait(&pipe->pipestatechange, &pipe->lock);
    int bytes_copied = fifo_copy_out(pipe, buffer, len);
    cond_signal(&pipe->pipestatechange, &pipe->lock);
    lock_release(&pipe->lock);
    return bytes_copied;
}
```

Alternative solutions are ok; for instance, modeling a pipe using two semaphores as discussed in class is possible as well – in this case, the counter in the semaphore could denote the number of elements in the buffer and the number of free slots, respectively.

3. Scheduling (20 pts)

- a) (8 pts) One of the new features in Microsoft's Vista operating system is "cycle-based CPU accounting." This method of CPU accounting uses the CPU's internal cycle counter register to accurately determine how much CPU time a thread consumed. The cycle counter register is incremented every clock cycle, so it can be used as a very accurate wall clock – for instance, on a 1GHz processor, it provides nanosecond accuracy.

The accounting works as follows: when a thread is scheduled, or before returning from an interrupt, the current value of the cycle counter is recorded as the thread's start time. When a thread is interrupted by an interrupt, or when the thread blocks, the cycle counter is read again and the delta is charged to the thread.

Consider that all versions of Windows use a variant of a MLFQS scheduler.

- i. (4 pts) Explain briefly why Vista's scheduler must know a thread's past CPU consumption!

General-purpose schedulers generally try to strike a balance between providing a SPN-like policy for I/O-bound processes with expected short CPU bursts, and

making sure that CPU-bound processes with expected long CPU bursts will not starve. Since general-purpose OS generally do not know the characteristics of a process beforehand, they try to infer it: in particular, they make the assumption that a process's future CPU consumption will be like its past consumption.

- ii. (4 pts) Name at least one problem with the approach used by the BSD 4.4 scheduler you implemented in project 1 that Vista's approach solves.

First, the approach does not sample like BSD's approach did. Therefore, it is not subject to the problems associated with sampling, such as being unable to discern frequencies higher than half the sampling rate – which is typical for I/O processes with short CPU bursts. Instead, CPU time is measured accurately by starting and stopping a clock.

Second, this approach accurately accounts for time spent in interrupts because a thread is not charged for this time: the thread's clock is stopped and then restarted while an interrupt occurs. Therefore, the conclusions drawn by the scheduler are more likely to be based on a thread's actual behavior, not on which interrupts happened to occur while it was running.

- b) (12 pts) Systems that use strict priority scheduling, such as the VxWorks Real-time Operating System used in the Mars Pathfinder mission, must implement some mechanism to avoid priority inversion.
 - i. (4 pts) Briefly define priority inversion.

Priority inversion occurs if a high-priority thread waits on a resource held by a low-priority thread, but the low-priority thread cannot make progress to release the resource because the scheduler picks a third, medium-priority thread instead. In this case, the relative priorities of the high and medium priority threads are inverted compared to the user's intention when assigning priorities to threads.

- ii. (4+4 pts) Can priority inversion also occur in general-purpose OS, such as Linux or Windows? If that is the case, why do those OS's not implement techniques such as priority donation to address this problem? If priority inversion does not occur, explain why not!

Priority inversion can certainly occur in general-purpose OS as well, there is nothing in its definition that would not apply to general-purpose OS.

Potential reasons why general-purpose OS do not commonly employ countermeasures include:

- *General-purpose generally do not use strict priority scheduling – instead, they employ a dynamic priority adjustment mechanism by which the*

dynamic, effective priority of the low-priority might eventually raised to be above the priority of the originally medium-priority thread, allowing it to make progress and release the resource the high-priority thread is waiting for.

- *The effect of a temporary inversion is probably not catastrophic, because unlike the Mars Pathfinder OS, general purpose OS do not provide real-time scheduling guarantees for their processes in the first place. Secondly, they also do not (frequently) employ watchdog timers that reset the system if a task is delayed.*
- *For completeness, we should point out that some general-purpose OS (in particular, Solaris 2), do implement priority inheritance on internal kernel locks.*

4. Deadlock (24 pts)

- a) (14 pts) One CS 3204 group implemented wait()/exit() using a semaphore that is used to signal a child process's completion to its parent, as follows:

```
process_wait(tid_t ctid)
{
    lock_acquire(process_list);
    find child c with tid == ctid in process_list
    sema_down(&c->exit_semaphore); // wait for child
    lock_release(process_list);
}

process_exit()
{
    lock_acquire(process_list);
    find own position o in process list
    sema_up(&o->exit_semaphore); // signal parent
    lock_release(process_list);
}
```

However, they found that their Pintos kernel hung for several tests, but not for all tests.

- i. (4 pts) Explain what sequence of events would lead to their kernel deadlocking and why.

If parent calls wait() before the child exits, the parent will acquire the process_list lock, and then block on sema_down (as the child has not yet exited). Now, when the child exits, process_exit() blocks on lock_acquire. This cyclic dependency leads to deadlock.

- ii. (4 pts) Explain under what circumstances deadlock would not occur.

No deadlock if the child exits before the parent calls `process_wait()`. Specifically, once the `process_exit()` acquires `process_list` lock, deadlock would not occur.

- iii. (6 pts) How could the code be fixed to avoid deadlock?

Moving `sema_up()` and `sema_down()` after `lock_release()` in both `process_wait()` and `process_exit()` is one solution to avoid a deadlock here. Other solutions that violate any one of the four conditions for deadlocks may also work.

- b) (10 pts) For this problem, consider locks only. One of the necessary conditions for a deadlock to occur is that there must be a cycle in the wait-for graph. To avoid such cycles, suppose you assigned a unique integer to each lock upon creation.

- i. (6 pts) Explain how you could change `lock_acquire()` to ensure that there will never be cycles in the wait-for graph!

The unique identifier could be used to enforce a strict order in which threads acquire multiple locks. This can be done by forcing the threads to acquire locks in a specific order (e.g. ascending order of integer IDs) only. If lock A has an identifier lower than lock B, `lock_acquire()` will allow threads to first get A and then B, but not vice versa. This prevents deadlocks due to different threads acquiring same lock in different order.

- ii. (4 pts) How would your technique affect the way the user would use `lock_acquire()`?

Previously, `lock_acquire()` either obtained the lock immediately and returned or it waited until it could acquire the lock. In either case, if it did return, the lock had been acquired.

With the proposed change, a call to `lock_acquire()` could now also fail if a thread tried to acquire locks in an order that violates the fixed ordering. Hence, the user would have to modify the calling code to handle this failure situation.