

Homework 2 solution

- Context switch between kernel-level threads A , B in the same process: make a system call (e.g. `yield`) or trap to the kernel, kernel scheduler chooses new thread B to run save the processor state on A 's stack, load processor state from B 's stack, reset timer to interrupt thread B after a time slice, return control to the application. Context switch between user-level threads: save processor state on A 's stack, load processor state from B 's stack. No system call or context-switch into the kernel is necessary, and no kernel concurrency control (e.g. interrupt disable, which cannot be directly invoked by user processes) is required, though user-level concurrency control is needed (this is usually done with atomic instructions).
- When the kernel makes a context-switch between two processes, it has to switch the active address space as well as the processor state. Since user processes resolve memory references using virtual addresses, this means that the page table pointer has to be changed, and the cache and TLB have to be flushed (since both store physical addresses). Though the actual time spent in the context-switch is not necessarily large, the new process will pay a penalty for restarting with a cold memory cache and TLB. Therefore, the performance of our hypothetical system could be improved by checking whether the "old thread" and "new thread" in the context switch share the same address space, and only flushing the cache and TLB if they differ.
- We'll assume Mesa-style condition variables:

```
monitor S {
    cond cv;      // condition variable for blocking
    int value;   // value of the semaphore

    void init(int v) {
        value = v;
    }

    void wait() {
        value--;
        if (value < 0)
            cv.wait();
    }

    void signal() {
        value++;
        if (value <= 0)
            cv.signal();
    }
}
```

- Suppose that N gives the number of chairs in the barber shop.

```
semaphore mutex = 1;      // mutual exclusion between customers
semaphore customers = 0; // counter of waiting customers
semaphore haircut = 0;   // customers wait to get haircut
int ncustomers = 0;      // new customer decides if it must wait
```

```

void barber() {
    for (;;) { // i.e. loop forever
        wait(customers);
        // "do haircut"
        signal(haircut);
    }
}

void customer() {
    wait(mutex);
    if (ncustomers < N) {
        ncustomers++;
        signal(customers);
        signal(mutex);
        wait(haircut);
        // "get haircut"
        wait(mutex);
        ncustomers--;
    }
    signal(mutex);
}

```

5. This problem is simplified considerably if we assume that when the agent puts down the two items, only the smoker who is missing exactly those two items picks them up (i.e. there is no competition between smokers for items).

```

semaphore smokers[3];

void smoker(int i) { // smoker 0 calls smoker(0), etc
    for (;;) {
        wait(smokers[i]);
        // smoke the cigarette
    }
}

void agent() {
    for (;;) {
        int i = random() % 3; // i.e. i one of 0, 1, 2
        signal(smokers[i]);
    }
}

```

6. This question becomes easier if you assume that `ExitBridge` can be called with a direction parameter. These sort of assumptions are fine if the question seems ambiguous and you explain the assumption you've made in your answer. Ask if you aren't sure whether an assumption is valid.

```

semaphore mutex = 1;
semaphore queued[2] = { 0, 0 }; // waiting to cross
int nqueued[2] = { 0, 0 };
int onbridge = 0; // always <= 3
int direction = 0; // who owns the bridge?

```

```

void ArriveBridge(int d) { // with direction
    wait(mutex);
    if (d != direction && onbridge == 0 && nqueued[direction] == 0)
        direction = d;
    if (d != direction) {
        nqueued[d]++;
        signal(mutex);
        wait(queued[d]);
        wait(mutex);
    }
    // d == direction now, and we have mutex
    while (onbridge == 3) {
        nqueued[d]++;
        signal(mutex);
        wait(queued[d]);
        wait(mutex);
    }
    onbridge++;
    signal(mutex);
}

void ExitBridge(int d) { // with direction
    wait(mutex);
    onbridge--;
    if (nqueued[d] > 0) {
        nqueued[d]--;
        signal(queued[d]);
    }
    else if (onbridge == 0 && nqueued[1-d] > 0) {
        direction = 1-d;
        while (nqueued[1-d] > 0) {
            nqueued[1-d]--;
            signal(queued[1-d]);
        }
    }
    signal(mutex);
}

```

7. In a multiprocessor computer, when a thread waits to acquire a lock, the thread which holds the lock could be running concurrently. Therefore, there might be an advantage to busy-waiting (i.e. spinning on the lock) for a while, in the hope that the holder will release it quickly. The alternative would be to block, incurring a delay of at least two context switches before being able to acquire the lock (one on the block, and one to resume running after being woken up by the lock holder). The disadvantage of spinning is that the lock holder may be in a long-running critical section, or may not be running concurrently. A reasonable amalgamation of spinlocks with blocking would be to spin to begin with, but to block if the lock has not been released within the duration of two context-switches. An even more sophisticated scheme might track the average length acquire-to-release interval for a lock and use this as a guideline for how long to spin before blocking.