

Solutions to Exam Two

CS160 - Operating Systems Drake University - Spring, 2004

Directions: This is an open book exam. Do all SIX problems. They have equal weight. Show all work. Please work first on problems with which you are more comfortable.

Problem 1. A heap is used to implement a priority ready queue. Assume that the heap is initially empty, and that the following events occur in the given order. Show *conceptually* (*i.e.* as a tree) what the heap looks like after each event.

1. Insert process 5 into the heap with priority 15.
2. Insert process 3 into the heap with priority 22.
3. Insert process 7 into the heap with priority 19.
4. Insert process 6 into the heap with priority 31.
5. Extract the process at the top of the heap.
6. Extract the process at the top of the heap.

Solution: (You were probably OK with this, so I'll skip it here.)

Problem 2. Explain briefly (but try to include all essential aspects) what Tannenbaum wishes to communicate in Figure 3-37 on page 252. Be sure to consider the different order in which events might occur, and how the system deals with this.

Solution: (You hopefully read this and/or followed it when we discussed it. You *should* be clear in indicating that there are two independent chains of events. What events initiate these? How and where do they come together?)

Problem 3. Consider what happens in Peterson's solution for achieving mutual exclusion (*i.e.* at most one process in a critical section at a time) when process 0 calls `enter_region(0)` at nearly the same time that process 1 calls `enter_region(1)`. Assume (although this isn't necessary) that the two processes share one CPU, and that preemption could occur at any moment. Describe two *significantly* different scenarios that might result, demonstrating that mutual exclusion is ensured in each case.

Solution: Assume one process starts executing `enter_region`. If it gets preempted by the other process before getting to the line that sets `interested[process]`, and if the preempting process does not get preempted before getting to the loop, then the loop condition will be false, and the preempting process will get into the critical section. When the first process starts running again, it will get caught in the loop until the other process gets out of its critical section. That's one of many possibilities. There are lots of other ways things could proceed. You just need to identify two of them, and show that things work out. For example, what if the first process gets preempted right after it sets `turn`, and the second process gets preempted just before reaching the line that sets `turn`. If the first process runs now, it will again get stuck in the loop. When the second process starts up again, it will set `turn` which will set the first process free, while the second gets stuck in the loop.

Problem 4. Besides the UP and DOWN operations on a semaphore, assume that an additional operation GET_VALUE can be employed to find out the current value of the semaphore.

A Volkswagen Beetle is used as transportation to get students from their dorm to I.H.O.P. for breakfast. It can hold up to twelve students (*not recommended in reality!!*), but there are many more students than this who have a craving for the current fad breakfast at I.H.O.P. Simulating students with processes, and using *SEMAPHORES* to coordinate them, write pseudo-code to do the following:

1. Allow students to attempt to get into the car;
2. Have students in the car actively repeat the message "There's still room." until the car is filled;
3. Try to arrange for the last student that gets into the car to become the driver, but in any case make certain that there is only one driver;
4. Have the driver call a `drive()` function, and have all others in the car call a `be_driven()` function, and be certain that nobody in the car is blocked (suspended).

My solution, which is more than I expected:

```

semaphore spots_left = 12;          /* No more that twelve people allowed in car */
semaphore mutex = 1;               /* Only one person can try to get in car at a time */
shared int number_in_car = 0;      /* This plus spot_left should always be twelve. */
shared int driver_pid = 0;         /* While zero, there is no driver */
shared int number_got_out = 12;    /* After each ride, will count until everybody leaves */
                                   /* Begin by pretending a ride just ended and everybody got out */

void main()
{
    int my_pid = getpid()           /* Get my PID */
    down(&spots_left);              /* Don't overfill car */

    /* Get in car. Increment number in car. Last one in becomes driver */
    down(&mutex);
    if (++number_in_car == 12 && driver_pid == 0)
        driver_pid = my_pid;
    up(&mutex);

    /* Busy waiting as requested in the problem - hang here till ready for ride */
    while (number_in_car < 12 || number_got_out < 12)
        if (number_in_car < 12) printf("There's still room!\n");

    /* Time to simulate the ride */
    if (driver_pid == my_pid)
        /* Code for driver */
        drive();                    /* Simulate driving somehow */
    else
        /* Code for passengers */
        be_driven();                /* Simulate being passenger somehow */

    /* I'm assuming that simulated ride is over when somebody gets here */
    down(&mutex);
    if (number_got_out == 12) number_got_out = 0;
    if (driver_pid == my_pid) {     /* Driver will be last one to get out */
        while (number_got_out < 11); /* Sorry - some busy waiting by driver */
        driver_pid = 0;
    }
    --number_in_car;                /* Get out of car when ride ends */
    ++number_got_out;
    up(&mutex);

    up(&spots_left);
}

```

Problem 5. Arnold is currently holding one widgets and might request up to three more widgets in the future. Betty is currently holding two widgets and might request just one more widget in the future. Carl is currently holding one widget and might request up to two more widgets in the future. Donna is currently holding three widget and might request up to six more widgets in the future. There is one widget still available. Is this situation safe (in the sense of potential deadlock)? Explain your answer in detail.

Solution: No, but almost. Here is how it could almost work. If anybody requests a widget other than Betty, then block them. If Betty requests a widget, let her have the last one. Whether or not she ever request this widget, Betty is sure to finish at some point and will then release any widgets she holds. At this point there will be three available widgets. Now if Arnold or Carl is blocked wait on one or more widget, or if either makes a request in the future, then the request can be filled. But if both of them want more widgets, then one of them must stay blocked until the other completes. Eventually both Arnold and Carl will be able to complete. At this point there will be five widgets available, and only Donna is left. But Donna might request six widgets, and this request could not be granted. In a sense, this isn't really "deadlock" since ultimately only one process is potentially going to cause trouble, but As long as you gave a sense of the thinking here, don't worry about the language.

Problem 6. The following table reflects the current usage of four types of resources among four processes. The notation m/n in the table means that a particular process might at some moment require up to n instances of a particular type of resource, and that it is currently holding m of these.

	$R1$	$R2$	$R3$	$R4$
$P1$	$1/5$	$1/3$	$2/3$	$1/3$
$P2$	$2/5$	$3/4$	$0/3$	$1/2$
$P3$	$1/3$	$2/4$	$2/3$	$0/2$
$P4$	$2/3$	$0/5$	$1/3$	$2/6$

Assume also that at the present time there are two instances of each type of resource still available. If $P1$ requests another instance of $R3$, will granting this request be safe, or could it result in deadlock. Explain your answer in detail.

Solution: Safe. After granting $P1$'s request, $P3$ could be allowed to finish, while blocking any other process that requests a resource. There are just enough available resources to guarantee that $P3$ can finish, even if it requests everything that it potentially wants. After this there will be three $R1$'s, four $R2$'s, three $R3$'s and two $R4$'s available. This allows $P2$ to finish (while blocking the others if necessary), which results in five $R1$'s, seven $R2$'s, three $R3$'s and three $R4$'s available. At this point, $P1$ can be allowed to finish, which results in six $R1$'s, eight $R2$'s, six $R3$'s and four $R4$'s available. Now $P4$ can finish.