VE482 - Homework 5

Introduction to Operating Systems

Kexuan Huang 518370910126

December 16, 2021

Ex.1 - Simple Questions

1. A system has two processes and three identical resources. Each process needs a maximum of two resources. Can a deadlock occur? Explain.

A deadlock can't occur. With 2 processes and 3 resources, one process could always get its maximum of 2 resources due to the pigeonhole principle. When one process finishes, the resouces it held would be released so that the other process could do its job. The condition of deadlock is not satisfied, since a process is not waiting for a resource while holding others.

2. A computer has six tape drives, with n processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?

n could be 1, 2, 3, 4 or 5. If n is 6 or more, considering the case that each process has one resource and is waiting for 1 more process to run to completion. And there are no more resources available as max 6 is reached. Then, a deadlock occurs. If we could have provided one more resource to any of the process, any of the process could have executed to completion, then released its resources, which further when assigned to other and then other would have broken the deadlock situation.

3. A real-time system has four periodic events with periods of 50, 100, 200, and 250 msec each. Suppose the four events require 35, 20, 10, and x msec of CPU time, respectively. What is the largest value x for which the system is schedulable?

The largest value is 12.5.

$$\frac{35}{50} + \frac{20}{100} + \frac{10}{200} + \frac{x}{250} < 1$$

$$x < 12.5$$

4. Round-robin schedulers normally maintain a list of all runnable processes, with each process oc- curring exactly once in the list. What would happen if a process occurred more than once in the list? Would there be any reason for allowing this?

If a process occurred more than once in the list, it will be granted more than one turn to run. The reason why this is allowed is that letting one process occurring more than once in the round-robin list can be treated as increasing the priority of that process. And it would be granted twice the CPU time (or more if it occurs third times) of the other processes.

5. Can a measure of whether a process is likely to be CPU bound or I/O bound be detected by analyzing the source code. How to determine it at runtime?

Yes. With source code, we can find that whether it is an I/O bound process which would usually require a number of read/write operations, or a CPU bound process which would usually be busy in computations. At runtime, commands like `top` and `iostat` could be taken advantaged of to see what types of the bound occurs.

Ex. 2 - Deadlocks

1. Determine the content of the Request matrix.

$$\text{The Request matrix} := \begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

2. Is the system in a safe state?

Yes. With available 332, we can run P_2 or P_4 first. After they are done, we have 743 available, so we can run P_1. Samilarly, P_3 and P_5 can be followed.

3. Can all the processes be completed without the system being in an unsafe state at any stage?

Yes, the processes can be completed without the system being in an unsafe state at any stage. The snapshot is shown below.

Process	Allocated	Maximum	Request	Available
				332
P_2	200	322	122	210
				532
P_4	211	222	011	521
				743
P_1	010	753	743	000
				753
P_3	302	902	600	153
				A55
P_5	002	433	431	624
				A57

Ex. 3 - Programming

Please refer to folder `banker` with more details.

Ex. 5 - The reader-writer problem

1. Explain how to get a read lock, and write the corresponding pseudocode.

To get a read lock, first, we should have a counter variable that counts the reader number. In the function `read_lock()`, we lock the reader counter. Then, we check whether it is the first reader. If it is, we lock the database. If it isn't, the database should already be locked, then we do nothing. After the reader count has been increased, we release the reader count.

```
size_t count = 0; // reader counter

void read_lock() {
  down(count_lock);
  if(++count == 1) down(db_lock);
  up(count_lock);
}

void read_unlock() {
  down(count_lock);
  if(--count == 0) up(db_lock);
  up(count_lock);
}
```

2. Describe what is happening if many readers request a lock.

If many readers are requesting a lock, they should wait for the reader counter to be released, call `read_call()`, access data in database and call `read_unlock()`. For the writer, they will wait until all readers finish to write to the database.

3. Explain how to implement this idea using another semaphore called read_lock.

When the writer lock the `read_lock`, other writers as well as other reader would be blocked until the writing process done and unlock it. In this way, instead of keep waiting until all the readers are done, the writer can write immediately after the previous job releases the lock.

```
void write_lock() {
  down(read_lock);
  down(db_lock);
}

void write_unlock() {
```

```
  up(db_lock);
  up(read_lock);
}

void read_lock() {
  down(read_lock);
  down(count_lock);
  if(++counter == 1) down(db_lock);
  up(count_lock);
  up(read_lock);
}

void read_unlock() {
  down(read_lock);
  down(count_lock);
  if(--counter == 0) up(db_lock);
  up(count_lock);
  up(read_lock);
}
```

4. Is this solution giving any unfair priority to the writer or the reader? Can the problem be considered as solved?

> This solution gives priority to the writer over the reader, while the previous one gives priority to the reader over the writer. As a result, this problem can't be consider as solved.