Hints or answers to Problem set 1

1. Introduction

1.2 The issue of resource utilization shows up in different forms in different types of operating systems. List what resources must be managed carefully in the following settings:
a. Mainframe or minicomputer systems
b. Workstations connected to servers
c. Mobile computers

Hint/Ans
a. Mainframes: memory and CPU resources, storage, I/O devices, network
b. Workstations: memory and CPU resources, network
c. Mobile computers: battery, screen, memory resources, network

1.8 What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?

Hint/Ans
hardware-generated vs. software-generated. a user program can.

1.9 Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU's execution load.
a. How does the CPU interface with the device to coordinate the transfer?
b. How does the CPU know when the memory operations are complete?
c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

Hint/Ans
a. through data register and control/status register
b. interrupt
c. in using the memory bus

1.12 Consider an SMP system similar to the one shown in Figure 1.6. Illustrate with an example how data residing in memory could in fact have a different value in each of the local caches.

Hint/Ans
Say processor 1 reads data A with value 5 from main memory into its local cache. Similarly, processor 2 reads data A into its local cache as well. Processor 1 then updates A to 10. However,

since A resides in processor 1's local cache, the update only occurs there and not in the local cache for processor 2.

2. System Structures

2.2 Describe three general methods for passing parameters to the operating system.

Hint/Ans
value in registers, starting address, use the stack

2.8 Why is the separation of mechanism and policy desirable?

Hint/Ans
Mechanism and policy must be separate to ensure that systems are easy to modify.

2.10 What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

Hint/Ans
See the textbook for advantages and disadvantages. User programs and system services interact by using the message passing technique that causes a serious performance problem.

2.13 Explain why Java programs running on Android systems do not use the standard Java API and virtual machine.

Hint/Ans
Standard ones are designed for desktop and server systems, not mobile systems.

3. Process Concept

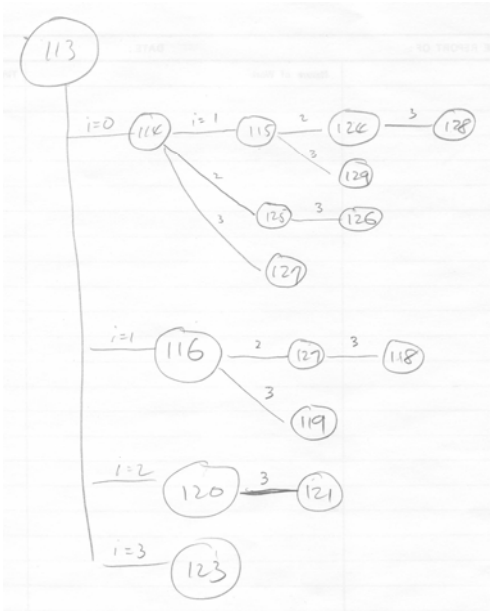
3.2 Describe the actions taken by a kernel to context-switch between processes.

Hint/Ans
See p. 112.

3.5 Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

Hint/Ans

16 processes. See the diagram.



3.6 Explain the circumstances under which the line of code marked printf("LINE J") in Figure 3.32 will be reached.

Hint/Ans
error in executing `exec( )`. why?

3.10 Using the program shown in Figure 3.34, explain what the output will be at lines X and Y.

Hint/Ans
the child at line X: 0, -1, -4, -9, -16. the parent at line Y: 0, 1, 2, 3, 4

3.11 What are the benefits and the disadvantages of each of the following? Consider both the system level and the programmer level.
a. Synchronous and asynchronous communication
b. Automatic and explicit buffering
c. Send by copy and send by reference
d. Fixed-sized and variable-sized messages

Hint/Ans

a. communication	synchronous	allows a rendezvous between the sender and receiver
	asynchronous	no wait → better performance and response
b. buffering	automatic	because of a queue with indefinite length, the sender will never have to block while waiting to copy a message. waste of memory possible.
	explicit	less likely to waste memory. may be blocked if no available space in the queue
c. send	by copy	may increase safety because the original copy thus sent cannot be corrupted. increased network traffic
	by reference	less traffic. may change the original in-place by the sender [benefit]. may be altered (less secure)
d. message size	fixed	simpler to manage the buffer and build the system. makes the task of [user] programming more difficult
	variable	makes programming easier (user level), but is more complex to build (system level). unable to know the number of messages to be held in the buffer

4. Multithreaded Programming

4.2 Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

Hint/Ans
frequent page faults or waiting for other system events

4.9 A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between startup and termination, the program is entirely CPU bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- How many threads will you create to perform the input and output? Explain.
- How many threads will you create for the CPU-intensive portion of the application? Explain.

Hint/Ans
- one. sequential I/O
- four. parallel computation

4.11 As described in Section 4.7.2, Linux does not distinguish between processes and threads. Instead, Linux treats both in the same way, allowing a task to be more akin to a process or a thread depending on the set of flags passed to the clone() system call. However, other operating systems, such as Windows, treat processes and threads differently. Typically, such systems use a notation in which the data structure for a process contains pointers to the separate threads belonging to the process. Contrast these two approaches for modeling processes and threads within the kernel.

Hint/Ans
scheduling point of view, as to distinction between processes and threads: for the Linux case, simple and one-step scheduling but difficult to impose process-wide resource constraints and figure out which threads belong to which process

4.12 The program shown in Figure 4.16 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

Hint/Ans
5, 0

5. Process Scheduling

5.1 Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?

Hint/Ans
I/O-bound programs: need less CPU time --> may not use up their CPU time quantum. if given higher priority, better use of CPU time

5.4 We have discussed possible race conditions on various kernel data structures. Most scheduling algorithms maintain a *run queue*, which lists processes eligible to run on a processor. On multicore systems, there are two general options: (1) each processing core has its own run queue, or (2) a single run queue is shared by all processing cores. What are the advantages and disadvantages of each of these approaches?

Hint/Ans
private queue for each core: [good] no contention over a common run queue (no concurrency control problem), simple scheduling decision, [bad] load balancing problem

5.8 The following processes are being scheduled using a preemptive, round-robin scheduling algorithm. Each process is assigned a numerical priority, with a higher number indicating a higher relative priority. In addition to the processes listed below, the system also has an *idle task* (which consumes no CPU resources and is identified as P_{idle}). This task has priority 0 and is scheduled whenever the system has no other available processes to run. The length of a time quantum is 10 units. If a process is preempted by a higher-priority process, the preempted process is placed at the end of the queue.

a. Show the scheduling order of the processes using a Gantt chart.
b. What is the turnaround time for each process?
c. What is the waiting time for each process?
d. What is the CPU utilization rate?

Hint/Ans
b. 20, 55, 60, 15, 20, 10
c. 0, 40(30?), 35, 0, 10, 0
d. 87.5%

5.11 Consider a variant of the RR scheduling algorithm in which the entries in the ready queue are pointers to the PCBs.

a. What would be the effect of putting two pointers to the same process in the ready queue?
b. What would be two major advantages and two disadvantages of this scheme?
c. How would you modify the basic RR algorithm to achieve the same effect without the

duplicate pointers?

Hint/Ans
a. raises the priority of the process
b. more important process gets more time while other processes suffer from less chance, can be implemented without changing the length of time quantum
c. allocate more time to higher priority processes. give two or more consecutive quanta to them

5.14 Consider a preemptive priority scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the CPU (in the ready queue, but not running), its priority changes at a rate α. When it is running, its priority changes at a rate β. All processes are given a priority of 0 when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.
a. What is the algorithm that results from $\beta > \alpha > 0$?
b. What is the algorithm that results from $\alpha < \beta < 0$?

Hint/Ans
a. FCFS b. LIFO

5.19 Assume that two tasks A and B are running on a Linux system. The nice values of A and B are−5 and +5, respectively. Using the CFS scheduler as a guide, describe how the respective values of vruntime vary between the two processes given each of the following scenarios:
• Both A and B are CPU-bound.
• A is I/O-bound, and B is CPU-bound.
• A is CPU-bound, and B is I/O-bound.

Hint/Ans
(1) Since A has a higher priority than B, vruntime will increase more slowly for A than B. If both A and B are CPU-bound, A will have a greater priority to run over B.
(2) vruntime will be much smaller for A than B (∵ pri(A) > pri(B) and I/O-bound processes need less CPU time)
(3) depends on how much time B would require for CPU in relation to A

5.20 Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

Hint/Ans
The priority inversion problem could be addressed by temporarily changing the priorities of the processes involved. Processes that are accessing shared resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When

finished, their priority reverts to its original value. This solution can be easily implemented within a proportional share scheduler; the shares of the high-priority processes are simply transferred to the lower-priority process for the duration when it is accessing the resources.

5.22 Consider two processes, P_1 and P_2, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t = 30$.
a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart such as the ones in Figure 5.16 – Figure 5.19.
b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

Hint/Ans
a. 25/50 +30/75 = ? ≤ or > U(2) = 0.828 ?
b. deadline = the end of period, schedulable

6. Synchronization

6.1 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: deposit(amount) and withdraw(amount). These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the withdraw() function and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Hint/Ans
Refer to the lecture notes and RUR notes
to prevent it from occurring: mutual exclusion mechanism

6.2 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 6.21. The other process is P_j (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

Hint/Ans
(1) Mutual exclusion is ensured through the use of the `flag` and `turn` variables.
(2) Use the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access the critical section, it can set the `flag` variable to true and enter the cs.

(3) Bounded waiting is preserved through the use of the `turn` variable. The algorithm has a process set the value of `turn` to the other process, thereby ensuring that the other process will enter its critical section next.

6.5 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Hint/Ans

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

6.9 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
int available;
} lock;
```

(available == 0) indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this struct, illustrate how the following functions can be implemented using the test_and_set() and compare_and_swap() instructions:

- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.

Hint/Ans

See Figures 6.4 and 6.6, and Section 6.5. Note that in this problem the setting of '`available`' was modified.

6.14 Consider the code example for allocating and releasing processes shown in Figure 6.23.

a. Identify the race condition(s).

b. Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

int number of processes = 0

with the atomic integer

atomic t number of processes = 0

to prevent the race condition(s)?

Hint/Ans

a. There is a race condition on the variable `number_of_processes`.

b. A call to acquire() must be placed upon entering each function and a call to release() immediately before exiting each function.
c. No, it would not help. The reason is because the race occurs in the allocate_process() function where number_of_processes is first tested in the if statement, yet is updated afterwards, based upon the value of the test.

6.17 Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the test_and_set() instruction. The solution should exhibit minimal busy waiting.

Hint/Ans

```
/* Initialize guard and semaphore_value */

wait()
{
      while (TestAndSet(&guard) == 1);
      if (semaphore_value == 0) {
            atomically add process to a queue of processes
            waiting for the semaphore and set guard to 0;
      }else {
            semaphore_value--;
            guard = 0;
      }
}

signal()
{
      while (TestAndSet(&guard) == 1);
      if (semaphore_value == 0 &&
            there is a process on the wait queue)
            wake up the first process in the queue
            of waiting processes
      else
            semaphore_value++;
      guard = 0;
}
```

6.20 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

Hint/Ans

```
monitor bounded_buffer {
      int items[MAX_ITEMS];
      int numItems = 0;
      condition full, empty;
```

```
      void produce(int v) {
            while (numItems == MAX_ITEMS) full.wait();
            items[numItems++] = v;
            empty.signal();
      }

      int consume() {
            int retVal;
            while (numItems == 0) empty.wait();
            retVal = items[--numItems];
            full.signal();
            return retVal;
      }
}
```

6.25 Consider a system consisting of processes P_1, P_2, ..., P_n, each of which has a unique priority number. Write a monitor that allocates three identical printers to these processes, using the priority numbers for deciding the order of allocation.

Hint/Ans

```
monitor printers {
      int num_avail = 3;
      int waiting_processes[MAX_PROCS];
      int num_waiting;
      condition c;

      void request_printer(int proc_number) {
            if (num_avail > 0) {
            num_avail--;
            return;
            }
            Waiting_processes[num_waiting] = proc_number;
            Num_waiting++;
            sort(waiting_processes);
            while (num_avail == 0 &&
                  waiting_processes[0] != proc_number)
            c.wait();
            waiting_processes[0] =
                  waiting_processes[num_waiting-1];
            num_waiting--;
            sort(waiting_processes);
            num_avail--;
      }

      void release_printer() {
            num_avail++;
            c.broadcast();
```

```
        }
}
```

6.28 Suppose we replace the wait() and signal() operations of monitors with a single construct await(B), where B is a general Boolean expression that causes the process executing it to wait until B becomes true.
a. Write a monitor using this scheme to implement the readers–writers problem.
b. Explain why, in general, this construct cannot be implemented efficiently.

Hint/Ans
a. The readers–writers problem could be modified with the following more generate `await` statements: A reader can perform "`await(active_writers == 0 && waiting_writers == 0)`" to check that there are no active writers and there are no waiting writers before it enters the critical section. The writer can perform a "`await(active_writers == 0 && active_readers == 0)`" check to ensure mutually exclusive access.
b. The system would have to check which one of the waiting threads have to be awakened by checking which one of their waiting conditions are satisfied after a signal. This requires considerable complexity and might require some interaction with the compiler to evaluate the conditions at different points in time.

- Priority inversion problem

Hint/Ans
A very simple synchronization protocol is :
- Always grant a lock request on a free semaphore.
- Execute the critical section at the priority of the task locking the semaphore.

Looking closer at the above example of synchronization, we can see what can go wrong with this simple protocol. In this example a low-priority task τ3 and a high-priority task τ1 share a semaphore. An intermediate-priority task τ2 does not need the semaphore.
- τ3 locks the semaphore and begins executing its critical section.
- Later, τ1 preempts τ3, executes for a while, and then tries to lock the semaphore.
- τ1 is blocked, because τ3 already has locked the semaphore; τ3 resumes execution.
- τ2 preempts τ3 and runs however long it wishes; τ1 is blocked for the entire duration.

The high-priority task τ1 is blocked for the duration of τ3's critical section, plus the entire duration of τ3's execution. Clearly, τ1 Is experiencing priority inversion. In fact, the duration of priority inversion is unbounded, since any intermediate-priority task that can preempt τ3 will indirectly block τ1.

7. Deadlocks

7.1 Consider the traffic deadlock depicted in Figure 7.10.
a. Show that the four necessary conditions for deadlock hold in this example.

b. State a simple rule for avoiding deadlocks in this system.

Hint/Ans
a. Illustrate how this example satisfies each condition.
b. One of simple rules is to leave the intersection unoccupied (passing is ok)

7.8 Consider a system consisting of m resources of the same type being shared by n processes. A process can request or release only one resource at a time. Show that the system is deadlock free if the following two conditions hold:
a. The maximum need of each process is between one resource and m resources.
b. The sum of all maximum needs is less than m + n.

Hint/Ans
See Section 7.6

7.9 Consider the version of the dining-philosophers problem in which the chopsticks are placed at the center of the table and any two of them can be used by a philosopher. Assume that requests for chopsticks are made one at a time. Describe a simple rule for determining whether a particular request can be satisfied without causing deadlock given the current allocation of chopsticks to philosophers.

Hint/Ans
When a philosopher makes a request for the first chopstick, do not grant the request if there is no other philosopher with two chopsticks and if there is only one chopstick remaining.

7.13 Consider the following snapshot of a system:
Answer the following questions using the banker's algorithm:
a. Illustrate that the system is in a safe state by demonstrating an order in which the processes may complete.
b. If a request from process P_1 arrives for (1, 1, 0, 0), can the request be granted immediately?
c. If a request from process P_4 arrives for (0, 0, 2, 0), can the request be granted immediately?

Hint/Ans
a. safe state: P1 - P0 - *
b. The requests are all additional to current state. See Section 7.5.3