CS330: Operating Systems
Mid-semester Examination Solution Sketch

Please be concise in your answers and write legibly.
Do not write unnecessarily long answers. Longer answers usually lead to lower scores.
This is an open-book/open-note examination.

1.(a) In both Linux and UNIX process schedulers, the scheduling priority of a process normally increases when the process goes to sleep due to the necessity to wait for certain events. As a result, the process receives a higher scheduling priority when it returns to the ready queue after completing the sleep. Cite two reasons supporting this particular priority policy. Assume that if the process goes to sleep because it called `sleep()`, its priority is not changed (otherwise this could be an easy way to boost one's priority). **(3+3 points)**

Solution: First reason: the process may be holding critical OS resources while sleeping. So, it must be allowed to free these resources as quickly as possible. However, it can free these only if it is scheduled. So, the process is given a priority boost. Note that the process is still executing in kernel mode. Second reason: interactive processes tend to sleep more due to more I/O activity. To improve the responsiveness of these processes, they are given a priority boost whenever they return from sleep.

1.(b) Let the effective CPU utilization be defined as the fraction of cycles the CPU spends executing instructions in the user mode. Briefly outline a situation that can lead to very low effective CPU utilization arising from the priority policy described in the previous question. What are the solutions adopted by the Linux and the UNIX process schedulers to remedy this situation? **(5+3+3 points)**

Solution: Consider a subset of processes in the system. Suppose these processes have frequent short I/O bursts involving blocking system calls and whenever they return from sleep they get a priority boost. As a result, these processes get scheduled more frequently than others. This can lead to a situation where the CPU spends more cycles in doing context switch than actually executing user mode instructions. The effective CPU utilization will be very low in such situations. The Linux OS has a starvation limit which puts an upper bound on the waiting time for a process in the ready queue. When this limit is crossed, such a process will receive a fairly high priority boost. This remedies the situation by balancing the mix of scheduled processes between CPU-heavy and interactive. The UNIX OS increases the priority of a process while it is waiting in the ready queue. So, any process waiting for too long will get the CPU soon. Both these solutions ultimately maintain the effective CPU utilization at an acceptable level.

1.(c) Consider a process scheduler that uses the round-robin scheduling policy with scheduling quantum τ. All processes have equal priority and the priorities do not change. However, the scheduler changes the value of τ dynamically based on the performance of the system. Outline an algorithm for adapting the value of τ. Your solution should clearly mention the steps in the algorithm, when the algorithm is invoked by the operating system, what metric the algorithm attempts to optimize and why optimizing such a metric is useful. **(10+2+5 points)**

Solution: My algorithm minimizes the number of preemptive context switches that a process undergoes while minimizing the maximum waiting time in the ready queue (i.e., ensuring an upper bound on the waiting time that a process spends in the ready queue between receiving two scheduling quanta). It is important to note that these two requirements are contradictory: the number of preemptive context switches that a process undergoes is inversely proportional to the scheduling quantum length and the maximum waiting time is directly proportional to the scheduling quantum length for a fixed set of processes. The algorithm is invoked at the end of each CPU burst and executes the following steps. The algorithm maintains, for each process, the length of the currently running CPU burst between two consecutive blocking system calls. It also maintains the last k CPU burst lengths observed.

- If the current CPU burst has ended due to expiry of scheduling quantum, increment the currently running CPU burst length of the currently running process by the scheduling quantum and skip the remaining steps. On the other hand, if the current CPU burst has ended due to a blocking system call, increment the currently running CPU burst length of the currently running process by the amount of time it has run during the current scheduling quantum, copy the updated running CPU burst length into a FIFO array of length k where k is a design parameter. Execute the following steps.
- Estimate the next CPU burst of the currently running process. Suppose that there are currently n active processes in the system. Take the list of the n estimated CPU bursts in the order of estimation and concatenate this list at the end of the last seen k CPU bursts.
- Compute maximum τ such that $(n-1)\tau$ is less than the upper bound on maximum waiting time. Note that $(n-1)\tau$ is the maximum waiting time that a process spends in the ready queue between receiving two scheduling quanta. Let us call this τ_0.
- Round down τ_0 to a multiple of timer interrupt interval. If τ_0 is bigger than the maximum CPU burst in the list of $k+n$ CPU bursts, set τ_0 to the maximum CPU burst in this list. Round down τ_0 to a multiple of timer interrupt interval. If τ_0 is less than the timer interrupt interval, set τ to timer interrupt interval; otherwise set τ to τ_0.

Notice that the algorithm automatically assigns a smaller τ to the scheduler when the number of processes increases. This is a desirable behavior because with an increasing number of processes unless the scheduling quantum is decreased, the waiting time for the processes at the tail of the ready queue is going to be very high.

1.(d) The turnaround time of a process is defined as the difference between the time the process is created and the time the process terminates. Express turnaround time as a sum of three mutually exclusive time components that typically a process goes through from its creation to termination. **(3 points)**

Solution: The turnaround time is the sum of CPU burst times, I/O burst times, and the waiting time in the ready queue.

1.(e) The minimum execution time of a process is defined as the time it takes to complete in a system when no other process is created or running. Under what condition the turnaround time of a process can be a cubic function of its minimum execution time? What should the nature of this function be in a good scheduler? **(3+3 points)**

Solution: The turnaround time is the sum of minimum execution time, waiting time in the ready queue, and waiting time in various event queues such that the occurrence of the events depends on the progress of the other processes in the system. The turnaround time of a process can be a cubic function of its minimum execution time, if the waiting time of the process in the ready queue is a cubic function of the minimum execution time.

Notice that even if we make the waiting times in the ready and event queues zero, the turnaround time of a process would be a linear function of its minimum execution time. This is the best one can achieve. A good scheduler should try to achieve the best and hence the nature of this function should be linear if the process scheduler is good.

2.(a) If a process chooses not to wait for the termination of its children i.e., it does not call `wait()`, it is recommended that the process inform the kernel through a `signal()` call that it will ignore the `SIGCHLD` signal. What is the rationale behind this recommendation? The `SIGCHLD` signal is used to notify the parent about the termination of a child process. **(3 points)**

Solution: If the process does not inform the kernel that it will not call `wait()`, the kernel will have to wait until the process terminates before it can free up the child process slots from the process table even if the children have terminated long back. This unnecessarily holds up the process table entries keeping track of zombie children. On the other hand, if the process informs the kernel that it will not call `wait()`, the kernel can free the process table entry of a child as soon as the child terminates.

2.(b) Consider two processes exchanging information through UNIX `msgsnd` and `msgrcv` calls. Can the processes use the same message type in their `msgsnd` calls or do they have to use different message types? Explain. **(3 points)**

Solution: Suppose both processes use the same message type for communicating. Consider one of the processes. Its `msgsnd` and `msgrcv` calls will use the same message type. As a result, it may end up receiving its own messages, since the `msgrcv` call matches messages using type alone. This was clearly not the intention. The processes must use different message types for communication.

2.(c) In UNIX System V, the signal handler field for signal N is reset to the default value zero before handling signal N. In Linux, this is not done. Explain the reasons for these different implementations in these two operating systems. What would be the outputs of the following program when executed on these two operating systems? Will the program terminate? Explain. **((3+3)+(3+3)+2 points)**

```
#include <signal.h>
#include <stdlib.h>
#include <sched.h>
#include <stdio.h>

#define CHILD_STACK 16384

int count = 0;

int loop (void *arg)
{
   while(1);
}

void catcher (int signo)
{
   count++;
}

int main (void)
{
   int i, pid;
   void *child_stack;
   for (i=1; i<=19; i++) {
      signal(i, catcher);
   }

   child_stack = malloc(CHILD_STACK);
   pid = clone (loop, child_stack+CHILD_STACK, SIGCHLD | CLONE_VM, NULL);

   for (i=0; i<5; i++) {
      kill (pid, SIGINT);
      sleep(10);
   }

   printf("Count=%d\n", count);
   kill (pid, SIGKILL);
   wait(NULL);
   return 0;
}
```

Solution: UNIX System V resets the signal handler field to avoid potential unbounded build-up of the process stack. However, this may lead to unexpected termination of the receiving process if another signal arrives before the handler field is set back. Linux avoids this problem by not resetting the signal handler field. To resolve the unbounded growth in process stack, the kernel blocks further receipt of signal N until the signal handler completes. This is possibly inherited from the BSD kernel. For the given program, UNIX System V will print 1 as the value of `count`, since the child will get terminated on the second signal. Linux will print 5 as the value of `count`, since the child gets terminated on `SIGKILL`. Note that `SIGKILL` terminates the receiving process immediately without invoking the signal handler. The program will terminate in both systems, since the child gets killed by either `SIGINT` (UNIX System V) or `SIGKILL` (Linux).

2.(d) What is the output of the following program? **(7 points)**

```
#include<setjmp.h>
#include <stdlib.h>
#include <stdio.h>

int g = 1;

main()
{
   jmp_buf env;
   int i, *p, a[10];
   register int k;

   for (i=0; i<10; i++) a[i] = i;
   p = (int*)malloc(sizeof(int));
   k = a[9];
   (*p) = 0;
   i=setjmp(env);
   printf("i=%d, a[5]=%d, k=%d, *p=%d, g=%d\n", i, a[5], k, *p, g);
   if (i==0) {
        for (i=0; i<10; i++) a[i]++;
        k++;
        (*p)++;
        g++;
    }
    else{
        exit(0);
    }
    printf("a[6]=%d\n", a[6]);
    longjmp(env,2);
    printf("a[7]=%d\n", a[7]);
}
```

Solution: No memory region is saved and restored as part of the context because those regions are guaranteed to be preserved. Only registers are saved and restored. The following two outputs are possible depending on whether `k` is allocated in a register. In the first output, `k` is in register, but in the second output, `k` is not in a register. The rest of the variables are always in memory e.g., `g` in global memory, `a` on stack, and `*p` on heap. Any of these outputs will receive full credit.

```
i=0, a[5]=5, k=9, *p=0, g=1
a[6]=7
i=2, a[5]=6, k=9, *p=1, g=2
```

```
i=0, a[5]=5, k=9, *p=0, g=1
a[6]=7
i=2, a[5]=6, k=10, *p=1, g=2
```