

Homework 1 - System Programming

Instructor for CS 241: Klara Nahrstedt
Deadline, March 6, 5pm
Department of Computer Science
University of Illinois, Urbana Champaign

The homework 1 is an individual effort activity!! (no pairs). Each student submits his/her own solution in the electronic form using the PDF format. The format should use 11pt or 12pt font. Each student submits his/her work to Illinois Compass in the same manner as he/she submitted machine problems:

1. Log into Illinois Compass (<http://compass.uiuc.edu>) and select cs241 from the *Course List*.
2. Click the *Add Attachments* button in the *Submissions* section.
3. In the *Choose Files* window, click *My Computer*.
4. Select your PDF document in the file selection dialog.
5. Click *Submit*, then check *OK* in the confirmation alert.
6. On the *Confirmation* page, click *Continue*.

1 Problems on Process/Thread Management(20 Points)

- (3 Points) If a system has only two processes, does it make sense to use a barrier to synchronize them? Why or why not?

Answer: If the program operates in phases and neither process may enter the next phase until both are finished with the current phase, it makes perfect sense to use a barrier. If the two processes are in producer/consumer relation, barrier synchronization mechanism is not appropriate. It is too expensive.

- (2 Points) In a system with threads, is there normally one stack per thread or one stack per process? Explain.

Answer: Each thread calls procedures on its own, so it must have its own stack for the local variables, return addresses, and so on. This is equally true for user-level threads as for kernel-level threads.

- (2 Points) What is a race condition?

Answer: A race condition is a situation in which two (or more) processes are about to perform some action. Depending on the exact timing, one or the other goes first. If one of the processes goes first, everything works, but if another one goes first, a fatal error occurs.

- (9 Points) What are the three main states that a process can be in? Describe the meaning of each one briefly as well as the relation among them (i.e., explain the transition conditions among these three states).

Answer: The three main states are *ready*, *running*, and *blocked* states.

A process is in the *ready* state if it can run once it would be selected by the scheduler, i.e., the process must wait in the ready queue to be selected again to run. A process gets into the ready state either by being temporarily stopped to let another process run, i.e., the time slice expired and it is moved from runnable to ready state, or by acquiring information/events it was blocked for, i.e., it is moved from blocked to ready state.

A process is in the *runnable* state if it holds the processor and executes instructions using the CPU. A process moves to runnable state if it was selected by the scheduler to run, i.e., it is moved from ready state to running state.

A process is in the *blocked* state if it is waits for some event to happen. For example, a process can wait on a semaphore to enter the critical region and it is blocked to wait for another process to free the semaphore and signal the blocked process. A process gets from the runnable state into the blocked state if (a) it wants to access critical region and waits for semaphore, (b) it executes I/O operation such as read or write operation that can be executed only by the kernel, (c) it waits for a signal (e.g., timer/alarm signal).

- (4 Points) Can two threads in the same process synchronize using a kernel semaphore if the threads are implemented by the kernel? What if they are implemented in user space? Assume that no threads in any other processes have access to the semaphore. Discuss your answer.

Answer: With kernel threads, a thread can block on a semaphore and the kernel can run some other thread in the same process. Consequently, there is no problem using semaphores.

With user-level threads and single threaded kernel, when one thread blocks on a semaphore, the kernel thinks the entire process is blocked and does not run it ever again. Consequently, the process fails.

2 Problem on Synchronization (20 Points)

Show how counting semaphores (i.e., semaphores that can hold an arbitrary value) can be implemented using only binary semaphores and ordinary machine instructions (you may use pseudo-code for the implementation). Explain the role of each semaphore that you use in the implementation of the counting semaphore.

Answer 1: S is a counting semaphore. S1, S2 and S3 are binary semaphores. S1 protects the counter variable C shared between wait(S) and signal(S); S2 protects S's critical region. S3 ensures sequential execution of wait.

```
S1 = 1
S2 = 1
S3 = 1
C = 0
```

```
wait(S) {
    wait(S3);
    wait(S1);
    --C;
    if (C < 0) {
        signal(S1);
        wait(S2);
    } else {
        signal(S1);
    }
    signal(S3);
}
```

```
signal(S) {
    wait(S1);
    ++C;
    if (C <= 0) {
        signal(S2);
    }
    signal(S1);
}
```

Answer 2: S is a counting semaphore. A and B are binary semaphores. B protects the counter variable C shared between wait(S) and signal(S). A protects S's critical region.

```
A=1;
B=1;
C=0;
```

```
wait(S) {
    wait(B);
```

```

C--;
if (C < 0) then
  { signal(B);
    wait(A); }
else
  signal(B);
}

```

```

signal(S) {
wait(B);
C++;
if (C<=0) the
  { signal(B);
    signal(A);}
else
  signal(B);
}

```

Another Solution 3: binary semaphore A protects the critical region, binary semaphore B protects the counter C, and the counter C counts how many processes want to access the critical region.

```

A=1;
B=1;
C=0;

```

```

wait(S) {
  wait(B);
  if (C <=0) then
  { C--;
    signal(B);
    wait(A);
  }
  else
  { C--;
    signal(B);}
}

```

```

signal(S) {
  wait(B);
  if (C < 0) then
    signal(A);
  C++;
  signal(B);
}

```

3 Problem on Synchronization (10 Points)

- (6 Points) Give a sketch of how an operating system that can disable interrupts could implement semaphores.

Answer: To do a semaphore operation, the operating system first disables interrupts. Then it reads the value of the semaphore.

If it is doing a **down** and the semaphore is equal to zero, it puts the calling process on a list of blocked processes associated with the semaphore.

If it is doing an **up**, it must check to see if any processes are blocked on the semaphore. If one or more processes are blocked, one of them is removed from the list of blocked processes and made runnable. When all these operations have been completed, interrupts can be enabled again.

- (4 Points) Does the busy waiting synchronization solution using the *turn* variable (see strict alternation solution, Figure 2-10 in Tanenbaum Book) work when the two processes are running on a shared-memory multiprocessor, that is, two CPUs, sharing a common memory? Why or why not?

Answer: Yes, the strict alternation solution works when two processes are running on a shared-memory multiprocessors. The processes still share the turn variable. But also in this case it is still a busy waiting which means that we waste two CPUs.

4 Problem on Scheduling (30 Points)

Consider the workload in Table 2:

Process	Burst Time	Priority	Arrival Time
P_1	50 ms	4	0 ms
P_2	20 ms	1	20 ms
P_3	100 ms	3	40 ms
P_4	40 ms	2	60 ms

Table 1: *Process Workload*

- (18 Points) Provide schedule using *preemptive Shortest Job First*, *non-preemptive Priority*, (a smaller priority number implies higher priority) and *Round Robin* with quantum 30 ms.

Answer is as follows:

- **preemptive Shortest Job First** (also known as *Shortest Remaining Time First*): The schedule is P_1, P_2, P_1, P_4, P_3 . Explanation: P_1 starts but is preempted after 20 ms when P_2 arrives and has shorter burst time (20 ms) than the remaining burst time of P_1 (30 ms). So P_1 is preempted. P_2 runs until the end. At 40 ms P_3 arrives, but it has longer burst time than P_1 , so P_1 will run. At 60 ms P_4 arrives. At this point P_1 has a remaining burst time of 10 ms, which is the shortest time, so it continues to run. Once P_1 finishes, P_4 starts to run since it has shorter burst time than P_3 .
 - **Non-preemptive Priority**: The schedule is P_1, P_2, P_4, P_3 . Explanation: P_1 starts, but as the scheduler is non-preemptive, it stays even though it has lower priority than P_2 . Once P_1 finishes, P_2 and P_3 arrives. Among these two, P_2 has higher priority, so P_2 will be scheduled and keeps the processor until it finishes. Now we have P_3 and P_4 in the ready queue. Among these two P_4 has higher priority, so it will be scheduled. After P_4 finishes, P_3 is scheduled to finish.
 - **Round-Robin**: The schedule is $P_1, P_2, P_1, P_3, P_4, P_3, P_4, P_3$. Explanation: P_1 arrives first, so it will get 30 ms quantum. After that P_2 already waits in the ready queue, so P_1 will be preempted and P_2 is scheduled for 20 ms. While P_2 is running, P_3 arrives. Note that P_3 will be queued after P_1 in the (FIFO) ready queue. So when P_2 is done, P_1 will be scheduled for the next quantum. It runs only 20 ms. In-between P_4 arrived, and it is queued after P_3 . So after P_1 is done, P_3 runs for 30 ms quantum. Once it is done P_4 runs for 30 ms quantum. Then again P_3 runs for 30 ms, after that P_4 runs for 10 ms and after that P_3 runs for 30 + 10 ms since there is nobody to compete with.
- (12 Points) What is the average waiting time of the above specified scheduling policies?
 - **Preemptive Shortest Job First**: The average waiting time is $\frac{20+70+10}{4} = 25$ ms. P_2 does not wait, but P_1 waits 20 ms, P_3 waits 70 ms and P_4 waits 10 ms.

- **Non-preemptive Priority:** The average waiting time is $\frac{30+10+70}{4} = 27.5$ ms. P_1 does not wait, P_2 waits 30 ms until P_1 finishes. P_4 waits only 10 ms since it arrived at 60 ms and it is scheduled at 70 ms. P_3 waits 70 ms.
- **Round-Robin:** The average waiting time is $\frac{20+10+70+70}{4} = 42.5$ ms. P_1 waits only for P_2 for 20 ms. P_2 waits only 10 ms until P_1 finishes the time slice (it arrives at 20 ms and the quantum is 30 ms). P_3 waits 30 ms to start, then 40 ms for P_4 to finish. P_4 waits 40 ms to start and one quantum slice for P_3 to finish.

5 Problems on Scheduling (20 Points)

- (8 Points) A soft real-time system, running Earliest Deadline Scheduling policy, has four periodic processes with periods of 50, 100, 200, and 250 msec each. Suppose that the four processes require 35, 20, 10, and x msec of CPU time, respectively. What is the largest value of x for which the system is schedulable?

Answer: The fraction of the CPU used is $35/50 + 20/100 + 10/200 + x/250$. To be schedulable, this sum must be less or equal to 1, i.e., x must be less than 12.5 msec.

- (12 Points) Five jobs are waiting to be run. Their expected run times are 9, 6, 3, 5, and X . In what order should they be run to minimize average response time? (your answer will depend on X). **Answer:** To minimize the average response time we should run Shortest Job First scheduling policy. For this, the following possibilities occur:

- if $0 < X \leq 3$ then schedule to minimize the response time is $X, 3, 5, 6, 9$;
- if $3 < X \leq 5$ then schedule to minimize the response time is $3, X, 5, 6, 9$;
- if $5 < X \leq 6$ then schedule to minimize the response time is $3, 5, X, 6, 9$;
- if $6 < X \leq 9$ then schedule to minimize the response time is $3, 5, 6, X, 9$;
- if $X > 9$ then schedule to minimize the response time is $3, 5, 6, 9, X$.