

CS423UG Homework 2 Solution

September 27, 2005

1. Let E be the CPU efficiency.

(a) $Q \rightarrow \infty$

In this case, round robin behaves exactly same as FCFS, i.e., once process possesses CPU, it finishes its job. Therefore, in average, CPU spends T seconds for serving one process, and S seconds for context switch, which gives us

$$E = \frac{T}{T + S}$$

(b) $Q > T$

Same as above. Since quantum is greater than the average process runtime, a process can finish its job within the quantum. Therefore,

$$E = \frac{T}{T + S}$$

(c) $S < Q < T$

We can expect $\lceil \frac{T}{Q} \rceil$ times of context switch for each process. Therefore,

$$E = \frac{T}{T + \lceil \frac{T}{Q} \rceil \cdot S}$$

Approximation can be made when $T \gg Q$, as follows.

$$E \approx \frac{1}{1 + \frac{S}{Q}} = \frac{Q}{Q + S}$$

(d) $Q = S$

The same argument applies to this case. Replacing Q with S gives

$$E = \frac{T}{T + \lceil \frac{T}{S} \rceil \cdot S}$$

If $T \gg Q$, we get $E = \frac{1}{2} = 0.5$.

(e) $Q \rightarrow 0$

In this case, most of the CPU time is used only for the context switch. Therefore $E \rightarrow 0$.

2. (a) Round Robin

Processes hold CPU in the following order, each for the quantum 0.1 sec:

$ABCDE \cdots ABCDE$: 1, 200times
 $ABDE \cdots ABDE$: 1, 200times
 $ABE \cdots ABE$: 1, 200times
 $AE \cdots AE$: 1, 200times
 $A \cdots A$: 1, 200times

Turnaround time for each process can be calculated as follows:

$$\begin{aligned}
 C : & \quad 1199 \cdot 0.5 + 0.3 & = 599.8 \\
 D : & \quad 1200 \cdot 0.5 + 1199 \cdot 0.4 + 0.3 & = 1079.9 \\
 B : & \quad 1200 \cdot 0.5 + 1200 \cdot 0.4 + 1199 \cdot 0.3 + 0.2 & = 1439.9 \\
 E : & \quad 1200 \cdot 0.5 + 1200 \cdot 0.4 + 1200 \cdot 0.3 + 1200 \cdot 0.2 & = 1680 \\
 A : & \quad 1200 \cdot (0.5 + 0.4 + 0.3 + 0.2 + 0.1) & = 1800
 \end{aligned}$$

Therefore, the mean turnaround time is:

$$\frac{599.8 + 1079.9 + 1439.9 + 1680 + 1800}{5} = 1319.92sec$$

(b) Priority Scheduling

Processes hold CPU in the following order, each for its length:

BEACD

Turnaround time for each process can be calculated as follows:

$$\begin{aligned}
 B : & \quad 6 & = 6 \\
 E : & \quad 6 + 8 & = 14 \\
 A : & \quad 6 + 8 + 10 & = 24 \\
 C : & \quad 6 + 8 + 10 + 2 & = 26 \\
 D : & \quad 6 + 8 + 10 + 2 + 4 & = 30
 \end{aligned}$$

Therefore, the mean turnaround time is:

$$\frac{6 + 14 + 24 + 26 + 30}{5} = 20min = 1200sec$$

(c) FCFS

Processes hold CPU in the following order, each for its length:

ABCDE

Turnaround time for each process can be calculated as follows:

$$\begin{aligned}
 A : & \quad 10 & = 10 \\
 B : & \quad 10 + 6 & = 16 \\
 C : & \quad 10 + 6 + 2 & = 18 \\
 D : & \quad 10 + 6 + 2 + 4 & = 22 \\
 E : & \quad 10 + 6 + 2 + 4 + 8 & = 30
 \end{aligned}$$

Therefore, the mean turnaround time is:

$$\frac{10 + 16 + 18 + 22 + 30}{5} = 19.2min = 1152sec$$

(d) SJF

Processes hold CPU in the following order, each for its length:

CDBEA

Turnaround time for each process can be calculated as follows:

$$\begin{array}{rcl} C : & 2 & = 2 \\ D : & 2 + 4 & = 6 \\ B : & 2 + 4 + 6 & = 12 \\ E : & 2 + 4 + 6 + 8 & = 20 \\ A : & 2 + 4 + 6 + 8 + 10 & = 30 \end{array}$$

Therefore, the mean turnaround time is:

$$\frac{2 + 6 + 12 + 20 + 30}{5} = 14min = 840sec$$

3. (a) FCFS

If short processes come first, FCFS performs efficiently because the overhead of context switch is minimal as well as they do not wait for CPU too much. However, if longer processes come first, FCFS performs worse because short processes cannot preempt the longer processes, and thus cannot be served by CPU until longer processes complete their jobs.

(b) RR

Since every process is given the same amount of time-slice regardless of its length, the turnaround time of a process depends mainly on its length and the number of processes. Therefore, short processes can finish their job relatively faster than longer processes.

(c) Priority

The similar explanation to FCFS can be made. If short processes have higher priority, Priority performs efficiently. However, if they have lower priority, Priority performs worse due to the same reason.

(d) Multilevel Feedback Queues

In multilevel feedback queues, the length of time-slice becomes bigger, as the level of queue goes deeper. Therefore, it is likely that short processes can finish their jobs within the first few levels, while longer processes end up with being placed on deeper-level queues.

4. • single-threaded

Assume that there are three requests. One of them involves a disk operation, and so a thread can process them in

$$3 \cdot 15 + 75 = 120ms$$

Therefore, in average, one request is processed in $120/3 = 40ms$, which gives us 25 requests/sec.

• multi-threaded

If you assumed that disk operations, even when overlapped take 75 ms, then your answer should be one request every 15 ms, or 66.67 requests/sec. If you assume that disk operations cannot be overlapped, then your answer should be 3 requests every 75 ms, or one request every 25 ms on average, which gives 40 requests/sec.

5. The following program is a possible solution.

```
typedef int semaphore;
semaphore mutex = 1;
semaphore tobacco = 0;
semaphore paper = 0;
semaphore matches = 0;

void agent(void)
{
    while (TRUE) {
        down(&mutex);
        Select random semaphore from tobacco, paper, or matches: call this <item>;
        Put missing ingredients on table;
        up(&<item>);
    }
}

void smoker(has tobacco)
{
    while (TRUE) {
        down(&tobacco);
        Get ingredients from table, make cigarette smoke;
        up(&mutex);
    }
}
```

Smoker who has paper or matches can be defined similarly. This solution avoids starvation.

6. The following is one possible solution.

```
semaphore mutex = 1; // accessing the sliding marker
semaphore menaccess = 1, womenaccess = 1; // wait for men/women
semaphore menwait = 0, womenwait = 0; // queue to enter the bathroom
int menwaiting = 0, womenwaiting = 0; // Boolean: are there men/women waiting to enter?
int mencount = 0, womencount = 0; // number of men/women inside bathroom
string sliding_marker = "empty"; // defined in problem

void Woman(void)
{
    down(womenaccess);
    down(mutex)
    if (sliding_marker == "men present")
        up(mutex);
        womenwaiting = 1;
        down(womenwait); // first waiting woman blocks - rest block on womenaccess above
        down(mutex);
    else if (sliding_marker == "empty")
        sliding_marker = "women present";
        womencount++;
        up(mutex);
}
```

```

up(womenaccess);

Use bathroom;

down(womenaccess);
down(mutex);
womencount--;
if (womencount == 0)          // sliding_marker=="women present"
    if (menwaiting == 1)
        sliding_marker = "men present";
        menwaiting = 0;
        up(menwait);        // wake up first waiting man
        else sliding_marker = "empty"
up(mutex)
up(womenaccess);
}

void Man(void)
{
    down(menaccess);
    down(mutex);
    if sliding_marker == "women present"
        up(mutex);
        menwaiting = 1;
        down(menwait);      // first waiting man blocks - rest block on menaccess above
        down(mutex);
    else if (sliding_marker == "empty")
        sliding_marker = "men present";
    mencount++;
    up(mutex);
    up(menaccess);

    Use bathroom;

    down(menaccess);
    down(mutex);
    mencount--;
    if (mencount == 0)          // sliding_marker=="men present"
        if (womenwaiting == 1)
            sliding_marker = "women present";
            womenwaiting = 0;
            up(womenwait);   // wake up first waiting woman
            else sliding_marker = "empty";
    up(mutex);
    up(menaccess);
}

```

The above solution(s) may cause starvation and/or deadlocks.

7. The following is a solution written in Pascal-type pseudo-code. We will assume that signal on a condition variable dequeues processes in FIFO order. We will assume Hansen-type condition variables (signalled process runs next, signalling process exits monitor immediately). Both procedures in the monitor are called by *processes* (not printers). This solution does NOT assume that processes have unique priorities or that a process can join only once. However, if each process uses the printer exactly once, the solution avoids starvation.

```
monitor Printers
  condition priority[1..n];
  int count[1..n];
  int printers_in_use;

  procedure acquire_printer(process: int);
  begin
    if printers_in_use = 3 then
      begin
        count[process] := count[process] + 1;
        wait(priority[process]);
      end
    printers_in_use := printers_in_use + 1;
  end;

  procedure release_printer(process: int);
  begin
    printers_in_use := printers_in_use - 1;
    for i := n to 1 do
      if count[i] > 0 then
        begin
          count[i] := count[i] - 1;
          signal(priority[i]);
          exit;
        end
      end
    end;

    for i:=1 to n do count[i] := 0;
    printers_in_use := 0;
  end monitor;
```

```

8. #define free 0
#define busy 1
semaphore mutex = 1;
semaphore calvin_cust = 0, terri_cust = 0, eddie_cust = 0;
semaphore calvin_barb = 0, terri_barb = 0, eddie_barb = 0;
semaphore chairs = 0;
int waiting = 0;

void customers(void)
{
    down(mutex);
    if (waiting>=N){
        up(mutex); /* exit the shop */
        exit;
    }
    else if (waiting>0){
        waiting++; /* wait in the chairs */
        up(mutex);
        down(chairs);
        checkbarberswithpriority();
    } /* no one is waiting - some barber may be free */
    else checkbarberswithpriority();
    else{ /* all barbers busy - take chair */
        waiting++; /* wait in the chairs */
        up(mutex);
        down(chairs);
        checkbarberswithpriority();
    }
}

void checkbarberswithpriority(){
    down(mutex);
    if (calvin==free){ /* check barbers in order of priority */
        up(calvin_cust); /* wake up barber */
        calvin=busy;
        up(mutex);
        down(calvin_barb); /* the customer waits for the barber */
        get_haircut();
    }
    else if (terri==free){
        up(terri_cust);
        terri=busy;
        up(mutex);
        down(terri_barb);
        get_haircut();
    }
    else if (eddie==free){
        up(eddie_cust);
        eddie=busy;
        up(mutex);
        down(eddie_barb);
    }
}

```

```
        get_haircut();
    }
}
```

Code for Calvin is as follows.

```
void calvin(void)
{
    while (TRUE) {
        down(mutex);
        if(waiting>0)
            up(chairs); /* wake up some customer */
        waiting--;
        calvin=free;
        up(mutex);
        down(calvin_cust); /* sleep. zzz. */
        down(mutex); /* barber woken up */
        up(calvin_barb); /* the barber gets the customer */
        up(mutex);
        cut_hair();
    }
}
```

Code for terri() and eddie() are similar. Caveat: the above solution might cause starvation - under some circumstances, a customer may leave the shop without being serviced.