HW2 Solution Set

1.(a) If both processes try to get into critical region, there are 6 statements must be executed:

(a1) $interested[0] = TRUE$	(b1) $interested[1] = TRUE$
(a2) $turn = 0$	(b2) $turn = 1$
(a3) while($turn == process$ and $interested[1] == TRUE$)	(b3) while($turn == process$ and $interested[0] == TRUE$)

- Case 1: Both processes called *enter_region* function.

  First, if process A went to step (a3) while process B still not yet executed (b1), then process A will break out of the while loop immediately because interested[1] was still $FALSE$. Thus A goes into the critical region first and B will be blocked by (b3) because $turn$ will be 1 and $interested[0]$ was $TRUE$ unless A leaves critical region. The same argument holds if B went to step (b3) before (a1) was executed.

  Second, if previous case does not hold, then both $interested[0]$ and $interested[1]$ are $TRUE$ before (a3) and (b3) were executed. Note $turn$ must be either 0 or 1, depends on which of (a2) and (a3) goes first. That one will leave the while loop after the $turn$ was set to the other process and use the critical region while the other one was blocked.

  Note that either case the second process will have access to the critical region right after the first one leaves because at that time $interested[other]$ will become $FALSE$ for the second process. So it will not get blocked forever.

- Case 2: Only one process called *enter_region* function.

  In this case it will have the access of critical region immediately since $interested[other]$ will be $FALSE$.

Theorem 1. *Peterson's algorithm satisfies four conditions of critical region management.*

Proof. The four criteria for critical region management will be:
Condition 1: No two processes could be in critial region simultaneously.
This follows from *Case 1*.
Condition 2: No assumption had been made about speeds or the number of CPUs.
This is true since every possible execution order of those 6 statements has been considered.
Condition 3: No outside running process could block other processes.
This is ture because the *interested* value of outside running process is always $FALSE$.
Condition 4: No process has to wait forever.
This follows from *Case 1* and *Case 2*. □

(b) Generalization of Peterson's Algorithm:

```
#define FALSE 0
#define TRUE 1
#define N x /* x:number of processes */

int turn[N-1];
int interested[N]; /* All values were initially FALSE */

int count(void)
{
      int cnt=0;
      for(int i=0;i≤N;i++){
            if(interested[i]){cnt++;}
      }
}

void enter_region(int process) /* process is 0..N-1 */
{
      interested[process]=TRUE; /* show that you are interested */
      for(int i=0;i≤count()-1;i++){
            turn[i]=process; /* set flag */
            while(turn[i]==process and i≤count()-1); /* null statement */
      }
}

void leave_region(int process) /* process: who is leaving */
{
      interested[process]=FALSE;
}
```

At any time assume k processes have called *enter_region* function, and m of them have set its *interested*[] to $TRUE$. Then for these m processes, the return value of *count* function will be greater or equal to m. Without loss of generality we may assume these m processes were numbered $0, 1, 2, ..., m-1$.

Lemma 1. *Process p could enter critical region only if all count function call returns m, i.e. no other process set its interested*[] *to* $TRUE$ *during this time, and* $p, turn[0], turn[1], ..., turn[m-2]$ *is a permutation of* $0, 1, ..., m-1$

Proof. If process p entered critical region, its i value must be $count()-2$ before the last loop increment. Thus the reason it have left the last while loop must be $turn[count()-2] \neq p$,

which means some other process already called $turn[i] = process$ for $i = count() - 2$. Similarly, the reason this process left the last while loop must be other process called $turn[i] = process$ for $i = count() - 3$, and so on. Note $turn[a] \neq turn[b]$ for any $a \neq b$ because if $turn[a] = turn[b]$ =process q for some $a > b$, then the loop index i of q would be a and it has no way to set $turn[b]$ again. So at least $count() - 1$ distinct processes was needed to unblock p. There are only $m - 1$ processes, however, already proceeded to the for loop while $count() \geq m$ all the time. This concludes $count() = m$ when p just got the access to the critical region. Also, since all $turn[]$ were distinct, $p, turn[0], turn[1], ..., turn[m-2]$ will be a permutation of $0, 1, ..., m-1$, which also means all other processes were blocked by distinct layer of while loop, and cannot execute further statements before p leave critical region or processes not yet set $interested[]$ do so. □

Theorem 2. *No two process could use the critical region at the same time.*

Proof. If one process, say p, was already using the region, it must have set its *interest* flag. If any process q also wanted to use it, by lemma it needs at least $m-1$ processes to unblock it while only $m-2$ available. (m processes exclude p and itself) □

Theorem 3. *No process has to wait forever.*

Proof. This is equivalent to show that if no process were in the critical region, there must be some process not yet blocked in while loop and have further statement to execute. If some process calling *enter_region* function but not yet set the *interested* flag, then this is the desired step which would be executed. Thus we may assume all k processes have set their $interested[]$ to $TURE$. Now the *count* function will always return k. By pigeon hole princeple, there exists p such that $turn[i] \neq p, i = 0, 1, ..., k-1$. Thus process p will not be blocked by while condition and has further statements to execute. □

Corollary 1. *Modified Peterson's Algorithm is correct.*

Proof. Here no assumption was made about speeds or the number of CPUs and *interested* flag will always be $FALSE$ for outside processes. Together with the above theorem this completes the proof. □

2. The reason that the state variable was set to $HUNGRY$ was there should be a flag telling other processes that "I need to be woke up!", so after the last one of its neighbors finishing eating, it called the test function and woke it up. Note this algorithm do have a problem, consider philosophers tried to get forks in the following order: $1, 2, 3, 1, 3, 1, 3, ...$, then there is chance that philosopher 2 never get woke up, which is called *starvation*.

3. If changing the order of commands, the condition in *test* function will always be $FALSE$, thus the $HUNGRY$ philosopher never got its chance to eat. Moreover, after the state of original philosopher became $THINK$, no body had ways to know that its neighbor is still $HUNGRY$.

4. Shortest Job First algorithm gives the minimum average response time. So the order should be:

$X, 3, 5, 6, 9$ if $X \leq 3$

$3, X, 5, 6, 9$ if $3 < X \leq 5$

$3, 5, X, 6, 9$ if $5 < X \leq 6$

$3, 5, 6, X, 9$ if $6 < X \leq 9$

$3, 5, 6, 9, X$ if $X > 9$

5.(a) It is in fact the process sharing algorithm, i.e., the quantum is infinitely close to zero such that it looks like every process has fair share of CPU.

So at the beginning, 5 processes are using CPU and the 2 minutes process actually needs 10 minutes to finish. After that, only 4 process are still running, and all of them already done 2 minutes work. So the finishing time of 4 minutes process would be $10+(4-2)*4 = 18$ minutes. Now only 3 processes are running and all have completed 4 minutes work, so the 6 minutes process would take $18 + (6-4) * 3 = 24$ minutes. Similarly, the 8 minutes process needs $24 + (8-6) * 2 = 28$ and 10 minutes process needs $28 + (10-8) * 1 = 30$ minutes. The answer would be $\frac{10+18+24+28+30}{5} = 22$ minutes.

(b) The running order will be $B(6) \rightarrow E(8) \rightarrow A(10) \rightarrow C(2) \rightarrow D(4)$.

Turnaround time will be $6, 14, 24, 26, 30$, respectively.

So the answer would be $\frac{6+14+24+26+30}{5} = 20$ minutes.

(c) The running order will be $A(10) \rightarrow B(6) \rightarrow C(2) \rightarrow D(4) \rightarrow E(8)$.

Turnaround time will be $10, 16, 18, 22, 30$, respectively.

So the answer would be $\frac{10+16+18+22+30}{5} = 19.2$ minutes.

(d) The running order will be $C(2) \rightarrow D(4) \rightarrow B(6) \rightarrow E(8) \rightarrow A(10)$.

Turnaround time will be $2, 6, 12, 20, 30$, respectively.

So the answer would be $\frac{2+6+12+20+30}{5} = 14$ minutes.

6. Answer $= \frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2} = \frac{40}{8} + \frac{20}{8} + \frac{40}{4} + \frac{15}{2} = 25$msec.