GROUP PROBLEMS

1.a. (see pdf files for the graphs)

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>t(i)</td>
<td></td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>TAU(i) alpha = 1/10</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10.1</td>
<td>10.29</td>
<td>10.56</td>
<td>10.9</td>
<td>11.31</td>
<td>11.58</td>
<td>11.72</td>
</tr>
<tr>
<td>TAU(i) alpha = 1/4</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10.25</td>
<td>10.68</td>
<td>11.26</td>
<td>11.94</td>
<td>12.71</td>
<td>13.03</td>
<td>13.02</td>
</tr>
<tr>
<td>TAU(i) alpha = 9/10</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10.9</td>
<td>11.89</td>
<td>12.88</td>
<td>13.88</td>
<td>14.88</td>
<td>14.08</td>
<td>13.1</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>t(i)</td>
<td></td>
<td>10</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>20</td>
<td>20</td>
<td>1</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>TAU(i) alpha = 1/4</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>6.5</td>
<td>9.87</td>
<td>12.4</td>
<td>9.55</td>
<td>19.66</td>
</tr>
<tr>
<td>TAU(i) alpha = 3/4</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>4</td>
<td>2.5</td>
<td>15.62</td>
<td>18.9</td>
<td>5.47</td>
<td>38.86</td>
</tr>
<tr>
<td>TAU(i) alpha = 9/10</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>2.8</td>
<td>2.08</td>
<td>18.2</td>
<td>19.82</td>
<td>2.88</td>
<td>45.28</td>
</tr>
</tbody>
</table>
1b. Acceptable answers include the following:

Mean error = $\frac{\sum |\text{TAU}(i) - t(i)|}{n}$ where n is number of iterations measured/predicted

Mean square error $\frac{\sum (\text{TAU}(i) - t(i))^2}{n}$ where n is the number of iterations measured/predicted

Others that take into account the difference between measured and predicted values as well as the number of iterations

2. **Efficiency of Scheduling Algorithm Implementation (20 pts)**

Just writing $O(n)$ lost you points if you did not clearly define the parameter n. Complexity is always defined in terms of parameters of the algorithm.

Assumptions: Let R be the number of processes in ready state. Let B be the number of processes in the blocked state.

**Solution 1**: Assume queues are not sorted.

One queue:
(i) $O(R+B)$ - search through all PCBs to find the ready PCB with highest priority

(ii) $O(R+B)$ - search queue to find the PCB whose I/O is done.
(iii) $O(1)$ - just change the status field of PCB (OS knows PCB of running process)

Two queues:
(i) $O(R)$ - search through PCBs in ready queue to find one with highest priority
(ii) $O(B)$ - search queue to find the PCB whose I/O is done.
(iii) $O(1)$ - unlink from one queue and link into other queue

Constant of proportionality is greater to change the links than to change status field, but still $O(1)$. Two queues is better since on average you search half of the list and two queues have shorter lists. With one queue, in a heavily I/O bound (CPU-bound) system, the scheduler would waste time searching through irrelevant entries.

**Solution 2**: Assume queues are sorted (ready queue by priority, blocked queue by I/O device numbers), you need to include the complexity of insertions into a sorted queue.
Assumption: ready processes sorted in order by priority precede blocked processes sorted in order by I/O device number.

One queue:
(i) \( O(1) \) to pick the first process in the combined queue (since ready processes precede blocked processes).
(ii) \( O(R+B) \) – search through all the ready processes since we don’t know where the blocked ones start, then through the blocked ones to find the one whose device number matches. First part of search, just check status bit. After you find the first blocked process, then check the I/O device field in the PCB for a match.
(iii) \( O(R+B) \) to insert in the right place by priority or by device number.

Two queues:
(i) \( O(1) \) to pick the first process in the ready queue.
(ii) \( O(B) \) to search through the blocked queue for the process matching the device whose I/O finished
(iii) \( O(R) \) or \( O(B) \) to insert into sorted list.

Two queues is better.
INDIVIDUAL PROBLEMS

3. Synchronization of Parent and Child Processes (20 pts)

Thanks to Nikki and Mike C. for contributing to this nice, clean solution!

Write pseudo-code for a parent process that forks off 100 processes such that up to 15 processes with odd PIDs can simultaneously be in critical section while up to 25 processes with even PIDs can simultaneously be in critical section. This constraint applies to both children and the parent process. Note that a process can find out its own PID using the getpid() kernel call.

```c
semaphore odd = 25;
semaphore even = 25;

int pid, mypid;

main() {
    for (I = 1; I<=100; I++) {
        pid = fork();
        if (pid == 0) {  /* a child */
            break;
        }  /* end of loop */
    /* Both parents and children try to enter c.s. */
        mypid = getpid();
        if (isodd(mypid))  wait(odd);  c.s. signal(odd);
        else if (iseven(mypid)) wait(even); cs; signal(even);
    }  /* end main */
```
5. **SJF-EA Schedule Implementation (20 pts)**

This solution simplifies the computation of CPU burst time by defining it as the time from when process is scheduled as running process to when it stops running on the CPU, regardless of whether it gave up the CPU voluntarily or involuntarily (e.g. requested I/O service or was interrupted).

New fields in PCB:
- `start_time` /* holds time at start of last burst
- `burst_time` /* holds last actual burst
- `new_pburst` /* holds next predicted burst

New procedure `Min_pburst(Ready_Queue)` /* selects the PCB whose `new_pburst` value is minimum

Assume `t(0) = TAU(0)` are initialized to some value when a process created.
Assume `alpha` is a system defined parameter, set as a global value.
Assume the `idle()` routine causes the CPU to idle until an interrupt occurs

```
scheduler() { /* I am a SJF Scheduler that uses exp. avg */
  if (called by a finished process) {
    /* put the now finished running process in the zombie queue */
    Remove_zombie(Running_Process);
    /* Get a new running process */
    Running_Process = Min_pburst(Ready_Queue);
    Running_Process->start_time = read_time();
    if (Running_Process != NIL ) Context_Switch (Running_Process, DELTA);
    else idle(); }

  else if (called by I/O routine) {
    /* measure most recent actual burst */
    Running_Process->burst_time = read_time() - Running_Process->start_time;
    /* compute new predicted burst using exp. avg. formula */
    Append(Running_Process, Wait_Queue);
    /* Get a new running process */
    Running_Process = Min_pburst(Ready_Queue);
    Running_Process->start_time = read_time();
    if (Running_Process != NIL ) Context_Switch (Running_Process, DELTA);
    else idle(); }
```
else if (called by I/O interrupt) {
    Running_Process->burst_time = read_time() - Running_Process->start_time; /* most recent burst fragment */
    /* compute new predicted burst using exp. avg. formula */
    Running_Process->new_pburst = alpha * Running_Process->burst_time +
        (1 - alpha) * Running_Process->new_pburst;
    PCB = Select(Wait_Queue, IO_Done_PID);
    Append(PCB, Ready_Queue);
    Running_Process = Min_pburst(Ready_Queue);
    Running_Process->start_time = read_time();
    if (Running_Process != NIL ) Context_Switch (Running_Process, DELTA);
    else idle();
}

} else if (called by clock interrupt) {
    Running_Process->burst_time = read_time() - Running_Process->start_time; /* most recent actual burst */
    /* compute new predicted burst using exp. avg. formula */
    Running_Process->new_pburst = alpha * Running_Process->burst_time +
        (1 - alpha) * Running_Process->new_pburst;
    Append(Running_Process, Ready_Queue);
    Running_Process = Min_pburst(Ready_Queue);
    Running_Process->start_time = read_time();
    if (Running_Process != NIL ) Context_Switch (Running_Process, DELTA);
    else idle();

} /* end of solution */