1. **[25 POINTS]** For parts (a) through (d), calculate the individual turnaround times and total wait times for processes P₁, P₂, P₃, and P₄ as described in the table below. For processes “arriving” at time 0, assume they are in process number order, i.e., P₁, P₂. More generally, if multiple processes are to be added to the ready queue at the same time, use process number order to break the tie.

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival time</th>
<th>CPU burst time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0 ms</td>
<td>13 ms</td>
</tr>
<tr>
<td>P₂</td>
<td>0 ms</td>
<td>9 ms</td>
</tr>
<tr>
<td>P₃</td>
<td>4 ms</td>
<td>3 ms</td>
</tr>
<tr>
<td>P₄</td>
<td>9 ms</td>
<td>5 ms</td>
</tr>
</tbody>
</table>

[0.5pts for each answer in the tables; including "ms" is not required]

a. **[4 points]** Apply the First-Come First-Served (FCFS) scheduling algorithm.

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround Time</th>
<th>Total Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>13 ms</td>
<td>0 ms</td>
</tr>
<tr>
<td>P₂</td>
<td>22 ms</td>
<td>13 ms</td>
</tr>
<tr>
<td>P₃</td>
<td>21 ms</td>
<td>18 ms</td>
</tr>
<tr>
<td>P₄</td>
<td>21 ms</td>
<td>16 ms</td>
</tr>
</tbody>
</table>

b. **[6 points]** Apply the Shortest Remaining Time (SRT) scheduling algorithm, i.e. a preemptive version of the Shortest Job First (SJF) algorithm. Also calculate the number of preemptions each process experiences.

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround Time</th>
<th>Total Wait Time</th>
<th>Preemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>30 ms</td>
<td>17 ms</td>
<td>0</td>
</tr>
<tr>
<td>P₂</td>
<td>12 ms</td>
<td>3 ms</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>3 ms</td>
<td>0 ms</td>
<td>0</td>
</tr>
<tr>
<td>P₄</td>
<td>8 ms</td>
<td>3 ms</td>
<td>0</td>
</tr>
</tbody>
</table>
c. **[6 points]** Apply the Round Robin (RR) scheduling algorithm with a time slice (i.e. quantum) of 2 milliseconds. Also calculate the number of preemptions each process experiences.

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround Time</th>
<th>Total Wait Time</th>
<th>Preemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>30 ms</td>
<td>17 ms</td>
<td>5 or 6</td>
</tr>
<tr>
<td>P₂</td>
<td>26 ms (or 27 ms)</td>
<td>17 ms (or 18 ms)</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>13 ms (or 9 ms)</td>
<td>10 ms (or 6 ms)</td>
<td>1</td>
</tr>
<tr>
<td>P₄</td>
<td>18 ms (or 17 ms)</td>
<td>13 ms (or 12 ms)</td>
<td>2</td>
</tr>
</tbody>
</table>

d. **[6 points]** Apply the Round Robin (RR) scheduling algorithm with a time slice (i.e. quantum) of 3 milliseconds. Also calculate the number of preemptions each process experiences.

<table>
<thead>
<tr>
<th>Process</th>
<th>Turnaround Time</th>
<th>Total Wait Time</th>
<th>Preemptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>30 ms</td>
<td>17 ms</td>
<td>4 (or 3)</td>
</tr>
<tr>
<td>P₂</td>
<td>24 ms</td>
<td>15 ms</td>
<td>2</td>
</tr>
<tr>
<td>P₃</td>
<td>8 ms</td>
<td>5 ms</td>
<td>0</td>
</tr>
<tr>
<td>P₄</td>
<td>20 ms (or 17 ms)</td>
<td>15 ms (or 12 ms)</td>
<td>1</td>
</tr>
</tbody>
</table>

e. **[3 points]** Given the table of process data on the previous page, if you increase the time slice from 3 ms in increments of 1 ms and apply the RR algorithm starting at time 0, at what point (i.e. what time slice value) does the RR algorithm degenerate into the FCFS algorithm?

**[3 pts]** 13 ms (or highest CPU burst time)
2. **[25 POINTS]** Given the following C program, answer the questions presented below. Assume that all system calls complete successfully. Further, assume that the parent process ID is 32, with any child processes numbered 64, 65, 66, etc.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    pid_t * x = (pid_t *)malloc( sizeof( pid_t ) );

    printf( "PARENT: %d\n", getpid() );
    pid = fork();
    printf( "PARENT: %d\n", pid );

    *x = pid * 100;
    if ( pid == 0 )
    {
        printf( "CHILD: !\n" );
        *x /= 10;
        printf( "CHILD: %d\n", *x );
    }
    else if ( pid > 0 )
    {
        printf( "PARENT: %d\n", *x );
        wait( NULL );  /* remove this line for part (b) below */
    }

    *x -= 1;
    printf( "PARENT: :-D %d\n", *x );
    free( x );
    return 0;
}
```

a. **[12 points]** Write the exact terminal output in the space below. If multiple outputs are possible, please succinctly describe all possibilities by using a diagram.

- **[1pt]** PARENT: 32
- **[1pt]** PARENT outputs:
  - **[1pt]** PARENT: 64
  - **[1pt]** PARENT: 6400
- **[2pts]** these PARENT and CHILD output lines are interleaved
- **[2pts]** the order of output lines
- **[1pt]** CHILD outputs:
  - **[1pt]** PARENT: 0
  - **[1pt]** CHILD: !
  - **[1pt]** CHILD: 0
  - **[1pt]** PARENT: :-D -1
- **[1pt]** PARENT outputs last:
  - **[1pt]** PARENT: :-D 6399

*go on to the next page*
b. [8 points] If the \texttt{wait()} system call is removed from the code on the previous page, what is the \textbf{exact} terminal output? Write in the space below. If multiple outputs are possible, please succinctly describe all possibilities by using a diagram.

\begin{itemize}
\item [0pt] PARENT: 32
\item PARENT outputs:
  \begin{itemize}
  \item [0pt] PARENT: 64
  \item [0pt] PARENT: 6400
  \item [4pts] PARENT: \texttt{:-D 6399}
  \end{itemize}
\item CHILD outputs:
  \begin{itemize}
  \item [0pt] PARENT: 0
  \item [0pt] CHILD: !
  \item [0pt] CHILD: 0
  \item [0pt] PARENT: \texttt{:-D -1}
  \end{itemize}
\end{itemize}

[c] [3 points] In the code above, if \texttt{pid_t} is actually an \texttt{unsigned short}, how many bytes are allocated by the \texttt{malloc()} system call?

[3pts] 2 bytes

d. [2 points] What would happen if the \texttt{free()} system call is removed from the code on the previous page?

[2pts] The memory allocated on the heap would not be freed, but that does not matter since the process terminates next.
3. [25 POINTS] Given the following C program, answer the questions presented below. Assume that all system calls complete successfully. Further, assume that the parent process ID is 64, with any child processes numbered 128, 129, 130, etc.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    close( 1 );

    int rc = open( "J.txt", O_WRONLY | O_CREAT | O_TRUNC, 0664 );
    pid_t pid = fork();
    printf( "[$$ %d]", pid );

    if ( pid == 0 )
    {
        fprintf( stderr, "[!]" );
        rc = dup2( 1, 2 );
        fprintf( stderr, "[? %d]", rc );
    }
    else if ( pid > 0 )
    {
        wait( NULL ); /* remove this line for part (c) below */
        fprintf( stderr, "[# %d]", pid );
    }

    return EXIT_SUCCESS;
}
```

a. [9 points] Write the exact terminal output in the space below. If multiple outputs are possible, please succinctly describe all possibilities by using a diagram.

```
3pts
vvvv
[!] [## 128]
^^^^  ^^^^  
3pts  3pts
```

```plaintext
3pts
vvvv
[!] [## 128]
^^^^  ^^^^  
```
b. [9 points] Write the exact contents of the J.txt file below. If multiple outputs are possible, please succinctly describe all possibilities by using a diagram.

2pts
vvvvvv
[? 2][$$ 0][$$ 128]

2pts 2pts ([2pts] last two segments appear in either order)

2pts
vvvv
[!!!][## 128]

2pts 2pts ([2pts] order could be swapped)

go on to the next page →
4. **[25 POINTS]** Given the following C program, answer questions (a) and (b) below. Assume that all system calls complete successfully. Further, assume that the parent process ID is 128, with any child processes numbered 256, 257, 258, etc.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int p[2];
    int rc = pipe( p );

    printf( "%d-%d-%d\n", getpid(), p[0], p[1] );

    rc = fork();
    if ( rc == 0 )
    {
        rc = write( p[1], "HAPPYSAINTPATRICK'SDAY!", 23 );
    }

    if ( rc > 0 )
    {
        int n = p[0] * p[0] - p[1];
        char * buffer = (char *)calloc( n + 1, sizeof( char ) );

        rc = read( p[0], buffer, n );
        printf( "%d-%s\n", getpid(), buffer );

        free( buffer );
    }

    return EXIT_SUCCESS;
}
```

a. **[9 points]** Write the exact terminal output in the space below. If multiple outputs are possible, please succinctly describe all possibilities by using a diagram.

[3pts] 128-3-4

PARENT:
[2pts] 128-HAPPY

CHILD:
[2pts] 256-SAINT

[1pt] PARENT and CHILD lines could be interleaved (swapped)
[1pt] PARENT could be SAINT instead of HAPPY and likewise for CHILD

b. **[4 points]** Why is the first argument of the `calloc()` call `n + 1` (instead of just `n`)?

[4pts] Extra byte is for the '\0' character (NULL or EOS terminator)

go on to the next page →
c. [12 points] Given the following C program, write the **exact** terminal output in the space below. As per usual, if multiple outputs are possible, please describe all possibilities by using a diagram. Assume that all system calls complete successfully. Further, assume that the parent process ID is 256, with any child processes numbered 512, 513, 514, etc.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    char b[24];
    int p[2];
    int rc = pipe( p );

    int pid = fork();

    if ( pid > 0 )
    {
        close( p[0] );
        rc = dup2( p[1], 1 );
    }

    printf( "198419841984" );
    fflush( NULL );

    if ( pid == 0 )
    {
        close( p[1] );
        rc = read( p[0], b, 8 );
        b[rc] = '\0';
        printf( "%d-%s\n", getpid(), b );
    }

    return EXIT_SUCCESS;
}
```

---

\[4\text{pts}\]

---

```
19841984198419841984512-19841984
^^^^^       ^^^^^^^
```

go on to the next page ➔
5. [EXTRA CREDIT +1] Write no more than 2 lines of code below to cause a segmentation violation and core dump.

```c
int main()
{
    char * x;       /* or other variations */
    *x = 'A';
}
```

all done 😊