Operating System Final Project - Report

1. Typescript for compilation

```
PS C:\Users\peggy\OS Checkpoint\OS-checkpoint-5> make clean
del *.hex *.ihx *.lnk *.lst *.map *.mem *.rel *.rst *.sym *.asm *.lk
PS C:\Users\peggy\OS Checkpoint\OS-checkpoint-5> make
sdcc -c  testparking.c
sdcc -c  preemptive.c
sdcc  -o testparking.hex testparking.rel preemptive.rel
```

2. Screenshots and explanation

- **delay(n) & now()**

I use a parameter `timer` to record the current time and a data structure to maintain the delay time of each thread. The delay and timer will be updated after eight timer0 interrupts (timer increases by one, and delay time of each thread decreases by one, respectively). Since timer0 interrupts when it overflows (0xFFFF) and the timer0 frequency is 1/12 times of the clock frequency of 8051 (12 MHz), i.e. the timer frequency is 1 MHz, hence, the time unit of my timer is

$$8 \times 0\text{xFFFF} \times \frac{1s}{1\text{MHz}} = 8 \times 65536\ \mu s = 524288\ \mu s \approx 0.5s$$

Here we have a maximum number of four thread running concurrently, and the thread yield to others at every time interrupt. Assuming the worst case that all thread finish delaying at the same time. All the thread should run for at least one time within 1-time unit. So, I let the timer increase by one after eight timer0 interrupts, which means the threads yield for 8 times, and each thread will dominate for at least one time. Hence, we can ensure the thread will claim the resource exactly between n time units and (n+0.5) time units after calling delay().

delay(n)	
	The function delay(n) is defined in macro. After calling delay, number of n time units will be assigned to the data structure indicating the remaining delay time. The delay time will be updated at every time unit, and delay() will return right after the delay time of the current thread becomes 0.
now()	
	Returns the current time on timer.

■ Thread Create and Termination

This part is for the creation and termination of each thread. The thread will claim its resource on creation, and yield to another existing thread when terminated.

<table>
<tr><td colspan="2">ThreadCreate()</td></tr>
<tr><td>

```
// check to see we have not reached the max #threads
if (threadCount >= MAXTHREADS) return -1;
// otherwise, find a thread ID that is not in use,
if      ((threadBitmap & 0x01) == 0x00) { threadId_new = 0; threadBitmap |= 1; }
else if ((threadBitmap & 0x02) == 0x00) { threadId_new = 1; threadBitmap |= 2; }
else if ((threadBitmap & 0x04) == 0x00) { threadId_new = 2; threadBitmap |= 4; }
else if ((threadBitmap & 0x08) == 0x00) { threadId_new = 3; threadBitmap |= 8; }

// increment #threads
threadCount += 1;
```

</td><td>If the number of threads is equal to or large than 4, then immediately returns -1, which is an invalid thread ID. If not, find the empty bitmap, and assign the new thread ID. The number of threads is also increased by one.</td></tr>
<tr><td>

```
tempSP = SP;
SP = 0x3F + 16 * threadId_new;

__asm \
    PUSH DPL \
    PUSH DPH \
__endasm;

__asm \
    CLR A \
    PUSH ACC \
    PUSH ACC \
    PUSH ACC \
    PUSH ACC \
__endasm;

PSW = threadId_new << 3;
__asm
    PUSH PSW \
__endasm;

savedSP[threadId_new] = SP;
SP = tempSP;
```

</td><td>Calculate the stack location of the new thread. Also, initialize the registers to 0 and push the registers.</td></tr>
<tr><td>

```
return threadId_new;
```

</td><td>At last, return the thread ID of the newly created thread.</td></tr>
<tr><td colspan="2">ThreadExit()</td></tr>
<tr><td>

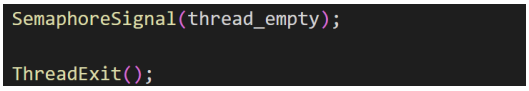
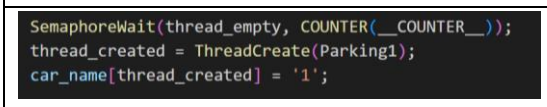
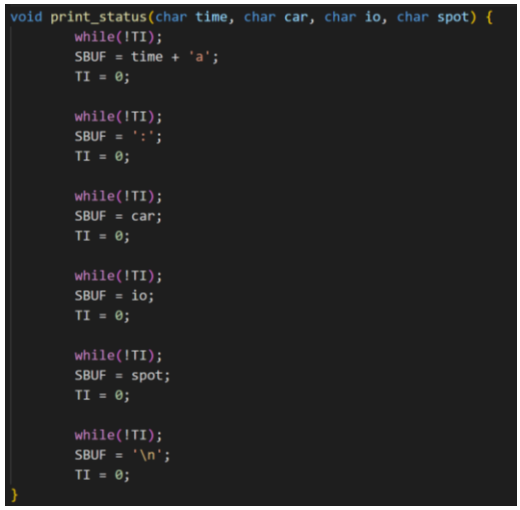
```
// TODO: clear the bit for the current thread from the bit mask
if      (threadId == 0) { threadBitmap -= 1; }
else if (threadId == 1) { threadBitmap -= 2; }
else if (threadId == 2) { threadBitmap -= 4; }
else if (threadId == 3) { threadBitmap -= 8; }

if (threadBitmap & 1) { threadId = 0; }
else if (threadBitmap & 2) { threadId = 1; }
else if (threadBitmap & 4) { threadId = 2; }
else if (threadBitmap & 8) { threadId = 3; }

else { while(1); }

RESTORESTATE;
```

</td><td>After calling ThreadExit(), the thread bitmap will be updated, and the current thread will yield to another existing thread. When there is no thread remaining, i.e. main() exits, it will enter an infinite loop.</td></tr>
</table>

■ Parking Lot Example

This is a implementation of parking 5 cars into 2 spots. The result is shown below. At time a (i.e. time 1), Car 1 enters Spot 0, and Car 2 entered Spot 1. After 3 time units, at time d (i.e. time 4), Car 1 and Car 2 both left the spots.

```
a:1i0
a:2i1
d:1o0
d:2o1
```

Parking()	
	Wait until there exists empty spot. Decide which spot is empty and put the car ID into the spot by assigning the car ID to the spot. After parking complete, print out the logs.
	Delays for a constant time unit of 3.
	After the delay time becomes 0, move out the car from its spot and reset the spot as empty. Again, output the log.
	When the thread finishes, signal the thread is empty. ThreadExit() will be called to yield the resources.
Make cars	
	In main(), the thread of the five cars will be created. A semaphore is added before calling ThreadCreate() to make sure that the total number of threads will not exceed four. The car ID of the assigned thread will be recorded.
Print logs	
	Finally, the log for the events are output to UART. I set the output format of time to be alphabets in order that the number is readable. For example, ‘a’ denotes the first time units, ‘b’ deontes the second, and so on. Then the following characters, ‘i’ or ‘o’, shows the car is entering or leaving the spot at Spot 0 or Spot 1.