COP 4634 Midterm Exam

Name Jonathan Harrod Grade 72 +8

1. True/False – Circle the correct response. Explain your answer if you feel it is necesary. (1 pts each) -28

T (F) 1. Global variables are stored in the process control block.

X (T) F 2. Interrupts must be disabled immediately after a hardware interrupt occurs.

(T) F 3. The mode bit changes from user to kernel mode when a system call is executed.

T (F) 4. Using the basic definition of a thread, stack is shared between two threads in a process.

X T (F) 5. User-level threads require less overhead in the execution of a context switch than kernel-level threads.

X (T) F 6. Using the basic definition of a process, data is shared between two processes.

(T) F 7. When an interrupt occurs, register values of a running process must be stored in its PCB.

T (F) 8. User-level threads are less vulnerable to priority inversion than kernel-level threads.

(T) F 9. A solution of the bounded buffer requires in addition to a lock two semaphores to control the producer and the consumer separately.

(T) F 10. The file descriptor table survives an `exec()` system call.

2. Fill in each blank with the correct word or phrase (2 pt each blank).

a) The scheduler selects a process from the Ready queue to assign it to the CPU for execution.

b) A system call triggers a software interrupt causing the process that makes the call to become blocked.

c) Mutual exclusion requires that a process or a thread acquires a lock before entering its critical section.

d) Elapsed time from when the system becomes aware of a new process to until the process first executes on the CPU is called response time.

e) Advanced scheduling algorithms prioritize I/O bound processes over CPU bound processes.

f) A busy loop is a loop that executes until a condition is met, but the condition can't be altered until the quantum expires and another process is scheduled.

g) A process shares data, code and heap with its threads.

h) The `yield()` system call is used to trigger a context switch causing the calling thread to abandon the CPU.

i) Actions that cannot be interrupted in the middle (either execute completely or not at all) are atomic.

j) Monitors implement ~~condition variables~~ mutual exclusion, preventing ~~access~~ race conditions on shared data.

-7

3. Complete the following pseudo code using semaphores to implement the corresponding conditional variable operations. Please note that Condition:: denotes that the `wait()` and `signal()` operation are part of a class or an abstract data type. (10 points)

```
Semaphore sem =    3   ;
Semaphore mutex = 1;
int       counter =  0   ;
```

```
void Condition::wait() {
    Disable_Interrupts();
    if(--counter < 0){
        waitlist.enqueue(self);
        Enable_Interrupts();
        block();
        Disable_Interrupts();
    }
    Enable_Interrupts();
}

void Condition::signal() {
    Disable_Interrupts();
    if(++counter <= 0){
        proc = waitList.dequeue();
        readyList.insert(proc);
    }
    Enable_Interrupts();
}
```

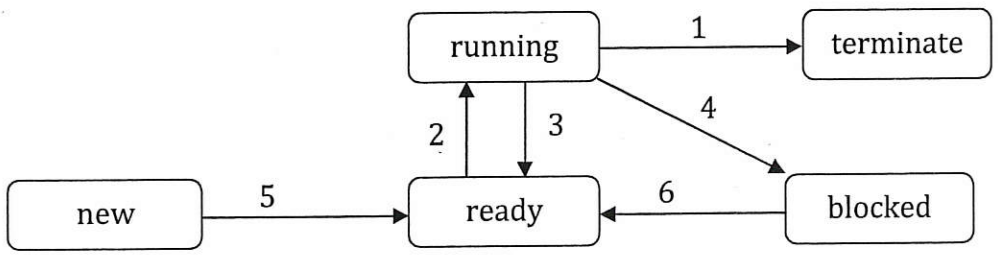
```
void Condition::broadcast() {
    Disable_Interrupts();
    while(counter!<0)
        signal();
    Enable_Interrupts();
}
```

```
mutex.wait()
counter++;
mutex.signal()
sem.wait()
```

```
mutex.wait()
if(counter>0){
    sem.signal()
    counter--;
}
mutex.signal()
```

```
mutex.wait()
while(counter >0){
    sem.signal()
    counter--;
mutex.signal()
```

4. Consider the process state transition diagram below.



Label each of the following transition descriptions with the corresponding number from the figure. (2 points each)

4	I/O is initiated	3	A quantum expires
6	I/O completes	2	A process is selected as next to run by scheduler
1	An exception occurs, the process returns from `main()`, or executes an `exit()`.		
5	Initialization of all internal data and allocation of needed resources is completed.		

5. Complete the **Response** and **Turnaround** times for each process in the following table. Show your work on the bottom of this page for partial credit. Start by drawing a **Gantt chart** for partial credit. Assume that a quantum is 6 ticks and a switch takes 1 tick where needed. Also assume, where needed, that arriving processes are placed on the tail of the ready list before any context switches take place. (12 points)

	t_a	CPU	FIFO				Round Robin				SRTCF			
			t_e	t_d	Resp	T/A	t_e	t_d	Resp	T/A	t_e	t_d	Resp	T/A
P_0	0	6	0	6	0	6	0	6	0	6	0	12	0	12
P_1	2	5	7	12	5	10	7	12	5	10	13	18	11	16
P_2	4	8	13	21	9	17	13	25	9	21	19	27	15	23
P_3	6	2	22	24	16	18	20	22	14	16	7	9	1	3

OK

FIFO

0 P_0
6 K
7 P_1
12 K
13 P_2
21 K
22 P_3
24 K

Round Robin

0 A_0^6
6 K
7 P_1^5 P_2^8 P_3^2
12 K
13 P_2^4
19 K P_2^2
20 P_3^2
22 K
23 P_2^2
25 K

SRTCF

- 0 P_0^6
2 K P_0^4 P_1^5
3 P_0^4
4 K P_0^3 P_1^5 P_2^8
5 P_0
6 K A_0^2 A_1^5 P_2^8 P_3^2
- 7 P_3^2
9 K
10 P_0^2
12 K
- 13 A_1^5
18 K
- 19 P_2^8
27 K

6. Consider the following program in C:

```
int a;
int main(int argc, char **argv) {
   int b = 20, pid;
   pid = fork();
   if (pid){
         a = 10;
         b = b + 10;
   } else {
         a = 50;
   }
   a = a - b;
   printf("%d :: %d :: %d\n", pid, a, b);
}
```

(1)

pid a b
5000 10 20
30
-20

pid_c a b
1000 50 20
30

a. List all possible outputs of this program assuming that the fork system call creates a child process with the process ID 1000. The parent's process ID is 5000. (6 points)

1000 :: 50 :: 30 OR 0 :: -40 :: 20
5000 :: -20 :: 30 1000 :: 30 :: 20

OR 1000 :: -20 :: 30 OR 1000 :: 10 :: 20 // child runs first, parent changes values before print (race condition)
0 :: 30 :: 30 5000 :: 10 :: 30

b. Explain what problem the program could create and then suggest a solution. (4 points)

Zombies

Race condition
Tell parent to wait() until child finishes

-2

c. Illustrate the execution of the program by completing the diagram below Think about variables that are on the stack and in the data section. (6 points)

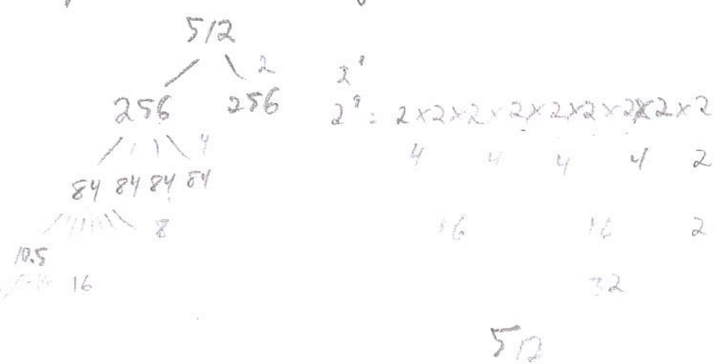
Process before fork()	Parent Process after completion of if-statement (1)	Child Process after completion of if-statement. (1)
Stack: b=20 ✓ pid=? ✓	Stack: b=30 ✓ pid=5000/? ✗	Stack: b=20 ✓ pid=1000/0
Data: a=? ✓	Data: a=10 ✓	Data: a=30 ✗
Code: <leave unspecified>	Code: <leave unspecified>	Code: <leave unspecified>
PCB: main() PID: ? ✗ 5000	PCB: main() PID: 5000 /? ✓	PCB: main() PID: 1000/0

-0.5 -0.5 -1

-9

7. Consider a program that implements parallel execution of data processing on an array of numbers based on the following scheme. For an array of $2*2^N$ numbers, processing is completed in N+1 stages. In the first stage, the array is divided equally into blocks of size 2 and each block is processed in parallel by a separate thread. In the second stage, the array is divided equally into blocks of size 4, each processed by a separate thread, and so forth. In the k-th stage, the array is divided equally into blocks of size 2^k each processed by a separate thread. Formally, thread T_i processes $i*2^k$ to $(i+1)*2^k-1$. Finally, in the last stage, the process combines results from the last two threads executing in parallel. Remember that in each stage, all threads have to completely terminate before the stage is complete and the program can proceed to the next stage. Complete the program below. (6 points)

completed in 9 stages



```
#define N 8
#define SIZE 512
#define T 256

int numbers[SIZE];

typedef struct {
   int from;
   int to;
} Range_T;

void* tFunc(void *val)
{
  int from = ((Range_T*)val)->from;
  int to = ((Range_T*)val)->to;

  // magic processing on array in locations ranging from 'from' up to 'to'
  pthread_exit(0);
}

int main(int argc, char** argv)
{
  int         i, j;
  int         n = T;      n=256
  int         step = 2;
  Range_T     range[T];
  pthread_t   tid[T];
                                         Range_T* start = range
  // parallel processing in N stages
  for (i=0; i < N; i++) {                X -1   missing range[] setup -1
        // start the threads
        for (j=0; j<SIZE   ; j++) {
              // select range
        -1 pthread_create(&tid[j], NULL, tFunc(SIZE), (void *)"T");
           range[j].from = j*step; range[j].to = (j+1)*step - 1;
                                             (void *) start++
        }
        // wait for the threads to complete
        for (j=0; j<SIZE+1 ; j++) {
           pthread_join("T", numbers[to][from]);   -1
                        tid[j], NULL
        }
        n = n / 2;
        step = step * 2;
  }
  // post processing (not to be completed by you)
}
```

-5

Over the run time of this program, how many threads in total does the program execute in parallel? (4 points)

2^9 - stages (n+1)

$2^9 = 512$ threads

X

510 -1

PThread API

```
pthread_create (pthread_t *thread,
                const pthread_attr_t *attr,
                void * (*start_routine) (void *),
                void *arg);
```

Example:

```
pthread_create (&tidA, NULL, tFunc, (void *)"threadA");
```

```
pthread_join(pthread_t thread, void **retval);
```

Example:

```
pthread_join(tidA, NULL);
```

4