Process Management Practice Exercises – Solutions

COMP-310/ECSE-427 Winter 23

Question 1: 6 times

Question 2: Only the shared memory segments are shared between the parent process and the newly forked child process. Copies of the stack and the heap are made for the newly created process.

Question 3: Even on a single CPU core, it still makes sense to launch multiple processes if it is known that some of the processes will do I/O. The reason is that some processes may be scheduled to run, while others are waiting for I/O, thus maximizing CPU utilization. Note that if too many processes are created, overhead due to context switching (i.e., scheduling) may become too large.

Question 4:

Client stub

```
int position(char c, char* s) {
   // no need to identify the RPC number, since the problem
   //statement tells us there's   only one RPC
  msg->arg0=c
  strcpy(msg->arg1, s) // s needs to be passed by value, not by reference.
  Send(msg)
  Receive(msg)
  pos = msg->retval0
return pos
}
```

Server stub

```
while(true) {
  Receive(msg)
  pos = position(msg->arg0, msg->arg1) //call server-side function
  msg->retval0 = pos
  Send(msg)
}
```

Message format

The message is a struct, containing the following fields:

- char arg0

- char* arg1 ← this field needs to be explicitly allocated in the message allocation helper function mentioned in the problem statement
- int retval0

Question 5:

1. 23 ms. See sketch below (black=CPU time, yellow=I/O time, gray= wait time).

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	⬛	⬛	⬛	⬛	🟨	🟨	⬛	⬛	⬛	⬛	🟨	🟨	⬛	⬛	⬛	⬛								
B	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	⬛	🟨	🟨	⬛	🟨	🟨	⬛	

2. 17 ms. See sketch below.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	⬛	⬛	⬛	⬛	🟨	🟨	⬛	⬛	⬛	⬛	🟨	🟨	⬛	⬛	⬛	⬛								
B	▨	▨	▨	▨	⬛	🟨	🟨	▨	▨	▨	⬛	🟨	🟨	▨	▨	▨	⬛							

3. 18 ms. See sketch below.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	⬛	⬛	▨	⬛	⬛	🟨	🟨	⬛	⬛	▨	⬛	⬛	🟨	🟨	⬛	⬛	⬛	⬛						
B	▨	▨	⬛	🟨	🟨	⬛	🟨	🟨	▨	⬛														

4.
Advantages of preemption
- Short job turnaround time is much better compared to non-preemptive (10ms vs 17ms)

Disadvantages of preemption
- Batch job (A) has longer turnaround time (18ms for preemption vs 16 for non-preemption)
- Here there is no context switching cost because we assume it is 0, but normally context switching overhead would be a disadvantage of preemption as well.
- Slightly longer execution time (18ms for preemption vs 17ms for non-preemption)
- More idle CPU time for preemption (3ms for preemption vs 2ms for non-preemption)

Question 6:
1. 10.53 time units
2. 9.53 time units
3. 6.86 time units

Question 7: A wait() operation atomically decrements the value associated with a semaphore. If two wait() operations are executed on a semaphore when its value is 1 and the operations are not performed atomically, then both operations might decrement the semaphore value, thereby violating mutual exclusion.

Question 8:

There are multiple solutions to the dining philosopher problem, and you are encouraged to explore several implementations. This solution uses the strategy where only 4 of the 5 philosophers are allowed at the table, as we saw in class. This implementation uses the C semaphore library which we have seen in class.
Note that in the final exam you will not be required to write code that compiles, but you may be asked to provide solutions in pseudo-code. The solution was adapted from here (link)

To compile the code on mimi, run: `gcc -pthread -o phil philosophers.c`

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<semaphore.h>
#include<unistd.h>

sem_t room;
sem_t chopstick[5];

void * philosopher(void *);
void eat(int);
int main() {
 int i,a[5];
 pthread_t tid[5];

 sem_init(&room,0,4);

 for(i=0;i<5;i++)
  sem_init(&chopstick[i],0,1);

  for(i=0;i<5;i++){
   a[i]=i;
         pthread_create(&tid[i],NULL,philosopher,(void *)&a[i]);
  }
  for(i=0;i<5;i++)
         // join will never happen
         // need to kill program with ctrl+c
         pthread_join(tid[i],NULL);
}

void * philosopher(void * num) {
 int phil=*(int *)num;

 while(1){
  sem_wait(&room);
  printf("\nPhilosopher %d has entered room",phil);
  sem_wait(&chopstick[phil]);
  sem_wait(&chopstick[(phil+1)%5]);
```

```
    eat(phil);
    sleep(2); // this could be a random value
    printf("\nPhilosopher %d has finished eating",phil);

    sem_post(&chopstick[(phil+1)%5]);
    sem_post(&chopstick[phil]);
    sem_post(&room);
  }
}

void eat(int phil) {
  printf("\nPhilosopher %d is eating",phil);
}
```