# SE 350 W23 Final Exam Solutions

**(1)**

**Part i.** proc2 frees proc1's memory. It doesn't run because of this.

proc1 and proc2 manage their own memory sharing (it is OK that memory is shared as long as there is always one and only one owner, as we posted on Piazza)

`p_msg->mtext` already is a string, so why would we ever cast like this?

We are wasting a memory block to transfer a string.

**Part ii.** The intention is to pass the string `"Hello, P2"` from proc1 to proc2, then have proc2 print it to the UART. both processes then end, and the NULL process and any other kernel processes remain the only running processes on the system.

**Part iii.**

```
void proc1(void)
{
    int     i = 0;
    int     j = 0;
    void    *p_blk;
    MSG_BUF *p_msg;
    char    *ptr;

    uart1_put_string("proc1: requesting a
        mem_blk...\r\n");
    p_blk = request_memory_block();
    p_msg = p_blk;
    p_msg->mtype = DEFAULT;
    ptr = p_msg->mtext;
    *ptr++ = 'H';
    *ptr++ = 'e';
    *ptr++ = 'l';
    *ptr++ = 'l';
    *ptr++ = 'o';
    *ptr++ = ',';
    *ptr++ = ' ';
    *ptr++ = 'P';
    *ptr++ = '2';

    send_message(PID_P2, p_blk);

    set_process_priority(PID_P1,LOW);
    while (1) {
        release_processor();
    }
}
```

```
void proc2(void)
{
    int i = 0;
    int j = 0;
    MSG_BUF *p_msg;
    void    *p_blk;

    uart1_put_string("proc2: receiving messages
        ...\r\n");
    p_blk = receive_message(NULL);
    p_msg = p_blk;

    uart1_put_string("proc2: got a message - ")
        ;
    uart1_put_string(p_msg->mtext);
    release_memory_block(p_blk);

    set_process_priority(PID_P2, LOW);
    while(1) {
        release_processor();
    }
}
```

**(2)**

```
int main(int argc, char **argv) {
  uint8_t childsum;
  uint8_t checksum;

  int shmid = shmget(IPC_PRIVATE, sizeof(size_t) +
      MAX_FILE_BYTES, IPC_CREAT | 0600);

  int pid = fork();
  if (pid < 0) { // fork error
    printf("Error␣occurred:␣pid␣=␣%d.\n", pid);
    return -1;
  } else if (pid == 0) { // child process
    // read file into shared mem
    FILE *file = fopen(argv[1], "r");
    void *mem = shmat(shmid, NULL, 0);
    size_t bytes = fread(mem+sizeof(size_t), 1, MAX_FILE_BYTES
        , file);
    fclose( file );

    *(size_t *)mem = bytes;
    checksum = calculate(mem + sizeof(size_t), bytes);
    shmdt(mem);

    return checksum;

  } else { // parent process
    // get child checksum
    int child_status;
    wait(&child_status);
    childsum = (uint8_t)WEXITSTATUS(child_status);

    // calculate checksum
    void *mem = shmat(shmid, NULL, 0);
    size_t bytes = *(size_t *)mem;
    checksum = calculate(mem + sizeof(size_t), bytes);

    // detach and destroy
    shmdt(mem);
    shmctl(shmid, IPC_RMID, NULL);

    // compare and return
    return childsum == checksum ? 0 : -1;
  }
}
```

- Open file - 1
- Attach shared mem segment - 1
- Read file into shared memory - 1
- Write size into shared memory - 1
- Detach shared memory (2x) - 1
- Child calculates and returns checksum - 1
- Parent waits for child and collects return value before proceeding - 2
- Parent retrieves size - 1
- Parent uses WEXITSTATUS - 1
- Parent calculates checksum - 1
- Parent destroyed shared mem - 1
- Close file - 1

**(3.1)**

1. **Hold-and-Wait**: If there's only one mutex it's not possible to hold the mutex and wait for another one because there is not second thing to wait for.
2. **Hold-and-Wait**: If we use trylock/trywait to acquire the constructs we won't get blocked even if unsuccessful, which removes the "wait" part.
3. **Hold-and-Wait**: If we intentionally forbid trying to acquire a second mutex we are sure we can't have hold-and-wait because we'll never wait for another one.
4. **Mutual Exclusion**: If there's no shared data, there's no need for using mutual exclusion at all!
5. **Cycle in Resource Allocation Graph**: If we always follow the ordering correctly then there is no way to build a cycle in the resource allocation graph. See what happens to the dining philosophers if they must acquire chopsticks in a certain order.

**(3.2)**

Your answer may vary, but here's what I'd say:

Tradeoffs: (1) the more conservative the algorithm is abut whether something should be allowed, the lower the performance of the app will be but less risk of deadlock; and (2) the more computation time this routine needs, the more accurate will be but the more it will decrease overall app performance.

Test scenarios: typical usage scenarios based on observed user behaviour. I want this, because I want to understand the impact on users. (You could also say performance testing).

Measures: I'd like to measure how effective it is in preventing deadlock, and what the cost in performance is.

**(3.3)**

Something like writing output (disk, network, console) may make it impossible to use rollback since we don't want to, for example, repeat a purchase. To address the problem you could either choose another deadlock recovery strategy, or defer such external operations until we're sure the transaction has succeeded.

**(4.1)**

The pointer tells us something about the number of cache misses in the system: if the pointer is moving quickly, it means there are a lot of misses so the system is probably operating slowly while it waits for data; if the pointer is moving slowly then it means the data needed is found in cache so it is likely the system is performing well.

**(4.2)**

$0.98 * 3ns + 0.02 * (0.9999 * 300ns + 0.0001 * 8ms) = 24.9394ns$

$0.98 * 3ns + 0.02 * (0.9999 * 300ns + 0.0001 * 70us) = 9.0794ns$

$24.9394/9.0794 = 2.747\times$ speedup.

**(4.3)** You could argue either yes or no but I'm going to argue yes. Guard pages do increase the amount of memory each process is using, and adds additional overhead for allocating it, configuring it, and deallocating it when it's no longer needed. However, it means that programs with bugs like writing past the end of an array are much more likely to consistently encounter a segmentation fault, and therefore it's more likely the bug gets fixed in development rather than causing arbitrary bad behaviour at runtime.

**(4.4)**

Part 1. The first 16 bits are the segment number, the next 12 bits are the page number, and the last 12 bits are the offset.

Part 2.

| **Virtual Address** | **Physical Address** | **Seg. Fault?** | **Page Fault?** |
|---|---|---|---|
| 0x00000030D1 | 0x40D1 | No | No |
| 0x00000001AC | N/A | No | Yes |
| 0x0001005B18 | N/A | Yes | No |
| 0x00010007FF | N/A | No | Yes |

**(5.1)**

**Round Robin:** $4 \to 5 \to 4 \to 2 \to 1 \to 3 \to 1 \to 3 \to 1 \to 3 \to 1 \to 3$

**Unequal Round Robin:** $4 \to 5 \to 2 \to 1 \to 3 \to 1 \to 3 \to 1 \to 3 \to 1 \to 3$

| **Process** | **Total Execution Time** | **Priority** | **Completion Time RR** | **Completion Time URR** |
|---|---|---|---|---|
| 1 | 4 | 3 | 18 | 18 |
| 2 | 7 | 2 | 11 | 11 |
| 3 | 10 | 3 | 25 | 25 |
| 4 | 3 | 4 | 4 | 3 |
| 5 | 1 | 1 | 2 | 4 |

Average completion time RR is 11.8; URR is 12.2 so it's slightly worse!

**(5.2)**

```
struct task {
  int          task_id;
  unsigned int priority;   /* lower values represent higher priority */
  int          ready;      /* a value of 0 means blocked (not ready);
                              any non-zero value means ready */
  /* ... additional data ... */
};

void dispatcher (struct scheduling_queue queues[], int num_queues) {
  struct task* current = get_current_task( );
  enqueue( queues[current->priority], current );
  struct task* next  = NULL;

  for (int i = 0; i < num_queues; ++i) {
      next = dequeue_ready_task( queues[i] );
      if ( next != NULL ) {
        break;
      }
  }
  dispatch( next );
}
```

Is this vulnerable to starvation? Yes! The algorithm always takes the highest-priority ready process so a process with very low priority may never be selected to run.

**(6.1)** Add item to buffer:

```
lock mutex
if current size + item size > capacity
  trigger empty
  current size = 0
add item to buffer
buffer size += item size
if buffer is full
  trigger empty
unlock mutex
```

Flush Buffer

```
lock mutex
if current size is not 0
  trigger empty
  current size = 0
unlock mutex
```

**(6.2)**

| **Algorithm** | **Service Order** | **Cylinders Moved** | **Improvement over FCFS** |
|---|---|---|---|
| First-Come First Serve (FCFS) | 365, 49, 350, 144, 434 | 1215 | 1.000 |
| Shortest Seek Time First (SSTF) | 434, 365, 350, 144, 49 | 418 | 2.901 |
| SSTF + Double Buffering, Buffer size 3 | 365, 350, 49, 144, 434 | 803 | 1.513 |
| SSTF + Double Buffering, Buffer size 4 | 365, 350, 49, 144, 434 | 803 | 1.513 |
| SCAN | (499) 434, 365, 350, 144, 49 | 482 | 2.520 |