

# Operating Systems – Spring 2005

## Test #1 – 2/18/2005

My name is: Answer Key

---

**Question 1 (15 pts):** This question involves the differences between threads and processes. Answer each part (5 pts each) with a brief paragraph (pictures are welcome).

- Describe 3 significant differences between a process and a thread.

Threads share code and heap (and global data), processes do not.

Threads share signals and alarms.

Threads can easily clobber each other (there is no protection between threads, there is some protection between separate processes).

If one thread calls exit, or crashes (SEGV), all the threads in the process die... Processes don't have this relationship.

Threads are much "lighter-weight", fewer system resources are needed to create a new thread than a new process (time and memory resources)

There are other differences... (give credit for anything that is significant!).

- Describe one application (situation) in which it is clear that it would be more appropriate to use multiple threads rather than multiple processes.

Since thread creation is much faster than process creation, an application that creates a new thread to handle small transaction would outperform the same application if it created a new process for each new transaction. A web server is a good example of such an application – each request is typically small (in terms of how long it takes to service one request), the overhead of creating a new process for each request is much greater than the overhead of creating a new thread.

Applications that require lots of communication between threads can share memory rather than using a pipe or other IPC mechanism (as would be needed with separate processes).

Many others are possible – anything that makes sense should get full credit.

- Describe one application (situation) in which it is clear that it would be more appropriate to use multiple processes rather than multiple threads.

A shell is a good example of an application that would do better with individual processes. As individual (external) commands are executed by the shell (like running the "ls" program), the shell remains alive as an independent process. If we were to try to do something like this with threads, each external command would need to start up a new shell upon completion (and this would change the nature of the shell entirely).

Other possible examples include applications in which it is important to isolate individual transactions from each other, so that if one screws up the others can continue working.

Full credit for anything that makes sense...

**Question 2 (25 pts):** You hear a rumor that you are limited in the total number of processes you can have running on monica.cs.rpi.edu at one time. You decide to write a program that will determine (and print out) this limit. Show the program below, try your best to write valid C code, but don't worry too much about perfect syntax (don't worry about include files, etc.) Make sure your code will convince us that you could write this program if you had a complete reference to all system calls. The last page of this test includes a list of a few Unix system call and library function prototypes.

```
/* General idea is to repeatedly fork until fork fails.
   The program must make sure that child processes don't
   exit until the limit has been reached - one way to do this
   is to have each process create one child, wait for that
   child to die, then exit. If fork ever returns an error we
   assume the limit has been reached and print out the number of
   processes that have been created. This doesn't take into
   account the shell process (or any others...)

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char **argv) {
    int f,stat,i=1;

    while (1) {
        if (f=fork()) {
            if (f<0) {
                printf("Maximum processes is about %d\n",i);
                exit(1);
            }
            /* parent just waits, then exits*/
            wait(&stat);
            exit(1);
        }
        i++;
    }
    return(0);
}
```

Grading notes: pseudo code is fine, they need to prove they understand how to use fork, how to make processes wait, etc.

They don't have to prove they remember all system call parameters...

**Question 3 (25 pts):** Consider the printer queue problem we discussed in class (this is basically a producer/consumer problem with many producers and a single consumer). There is a printer daemon that watches a print queue for jobs, and as it finds print jobs it sends them to the printer. Assume the queue has room to hold 10 print jobs at most. The producers are applications that create print jobs and want to submit them to the queue. For this question you can treat the queue as a shared array of *print jobs* (we are not concerned with the details of what a print job is).

Your job is to write the code (pseudo-code or C code) that controls the printer daemon (the consumer) and the applications (the producers). Your solution must use semaphores (with operations up() and down()) and avoid race conditions, deadlock and starvation.

**Note:** You need to use semaphores for this (not message passing, sleep/wakeup, etc). Recall that a semaphore can be used to make a mutex, and that there is no operation to “read” the value of a semaphore.

This is basically the same problem we went over in class (producer consumer problem using semaphores). To do this you need a semaphore to keep track of the empty slots, and another to keep track of the number of jobs in the queue. The consumer will block waiting until there is at least one job, and the producers will block waiting until there is an available slot.

```
#define MAXJOB 10
typedef int semaphore;
/* need a semaphore to keep track of number of jobs in the queue. */
semaphore curjobs=0;
/* need a semaphore to keep track of number of empty slots */
semaphore slotsfree=MAXJOB;
/* need a mutex to control access to the queue */
semaphore mutex = 1;
/* application is run by the producers of print jobs */

void application() {
    while (TRUE) {
        generate_print_job();
        down(&slotsfree);    /* wait until there is a free slot */
        down(&mutex);        /* enter critical region */
        submit_job();
        up(&mutex);          /* leave critical region */
        up(&curjobs);        /* update curjobs */
    }
}

void printer_daemon() {
    while (TRUE) {
        down(&curjobs);      /* wait until there is at
                             least one job in the queue */
        down(&mutex);        /* enter critical region */
        get_print_job_and_send_to_printer();
        up(&mutex);          /* leave critical region */
        up(&slotsfree);     /* update slotsfree */
    }
}
```

**Question 4 (16 pts):** Six *batch* jobs arrive at a computer center at almost the same time. They have estimated running times and priorities shown below. For each of the following scheduling algorithms, determine the mean process turnaround time (average response time). **Note:** Priority 6 is the highest priority, 1 is the lowest.

Job	Estimated Run Time	Priority
A	10 minutes	3
B	4 minutes	2
C	4 minutes	1
D	6 minutes	6
E	8 minutes	4
F	2 minutes	5

**Round-Robin** (the quantum time is not important, so assume a quantum of 1 millisecond).

Order is not relevant, assuming A,B,C,D,E,F Initially each process gets  $1/6^{\text{th}}$  of each CPU minute.  
 After 12 minutes, F completes (leaving the other 5 still running – each gets  $1/5$  of each CPU minute).  
 B & C both finish 10 minutes later (at time 22). This leaves D,E and A each getting  $1/3$  of each minute.  
 D finishes six minutes later (at time 28), leaving E and A, each getting  $1/2$ .  
 E finishes four minutes later (at time 32). Then F at time 34.  
 Average is 25 minutes.

**Priority scheduling.**

Order will be: D,F,E,A,B,C  
 turnaround times:  
 D: 6, F: 8, E: 16, A: 26, B: 30, C: 34  
 Average is 20 minutes

**First-come, First-served** (run in order A, B, C, D, E, F).

turnaround times:  
 A: 10, B: 14, C: 18, D: 24, E: 32, F: 34.  
 Average is 22 minutes

**Shortest Job First.**

Order will be: F, B, C, D, E, A  
 turnaround times:  
 F: 2, B: 6, C: 10, D: 16, E: 24, A: 34  
 Average is 15.3 minutes

**Question 5a (5pts):** Explain the differences between user-level threads and kernel-level threads. Include some discussion of the advantages/disadvantages of each.

Kernel threads are created and scheduled by the kernel. The kernel allows individual threads to block independently of each other. The kernel manages the distribution of CPU time to each thread, and in a multi-processor environment it is possible to have multiple threads running on different CPUs.

User level threads are based on a library that manages threads within a single process. The kernel is not aware of the individual threads, and so it only handles scheduling of the process. If any single thread makes a blocking system call, the entire process will block (other threads in the process can't run until the system call returns).

User threads requires less overhead, as the kernel doesn't need to be contacted to switch threads (saving a context switch). However, user threads can't take advantage of multiple processors or even hyperthreading.

**Question 5b (4 pts):** Describe some of the major differences between systems programming under Windows (win32 API) and Unix (Posix) that deal with process creation and running a new program.

Under Unix there are different system calls for i) creating a new process, and ii) running a new program. A new process is created with `fork()`, and an existing process runs a new program by calling one of the `exec` system calls. Processes form a hierarchy, each process has a parent process. Unix also supports the notion of process groups.

Windows does not support a process hierarchy – each process is on it's own. There is a single system call that creates a new process and starts it running a new program: `CreateProcess()`.

When a new process is created under Unix, many process attributes are inherited from the parent (user id, working directory, etc) – under Windows most attributes are specified via parameters to the `CreateProcess` system call.

**Question 6 (10 pts):** Consider what happens when you type in a command to a Unix shell – the shell looks at the command name and attempts to find a file with that name in your *path*. The *path* is an ordered sequence of directory names that should be searched. The first match (first file found in one of the directories in the *path* that is an executable file) is then executed. For example, if your path contains the directories `/usr/bin`, `/usr/sbin`, `/usr/local/bin` and the shell is given the command name “ls”, it looks for `/usr/bin/ls`, then `/usr/sbin/ls`, and finally `/usr/local/bin/ls`. Your job is to write this code (pseudo-code is fine, but use real Unix system call names where appropriate). Your pseudo-code should find the file that corresponds to the command name (if it exists), start up a new process and use it to execute the command. You should package this code as a function named `execute_command` with the prototype:

```
int execute_command(char *cmd, char **path);
```

- The parameter `cmd` is a pointer to a string containing the command name. You don't have to worry about passing any command line options to the command.
- The parameter `path` is pointer to an array of `char *`, each is a pointer to a string containing a directory in the path. The array `path` is terminated by a NULL pointer.
- The function should return 1 if it finds and the command in the path (and executes the command), 0 if there is a problem.

```
int execute_command( char *cmd, char **path) {
```

[Actual C code included on next page.](#)

The general idea is to iterate through the components of the path, for each one the function creates a pathname by appending the command to the path. Then calls `stat` on this path and if `stat` finds a file (that is a regular file and executable), the shell then forks. The parent waits for the child to finish, the child execs the program.

pseudo-code is fine (expected!). They should be using actual system call names, but they don't need to write C code. Comments like “of course we need to check for errors” should be considered as error checking everywhere...

This is only worth 10 points! Give them most of the points if they have the right idea, don't be picky about syntax, etc.

```

int execute_command(char *cmd, char **path) {
    int i;
    char buf[MAX_PATH];
    struct stat sb;

    while (*path != NULL) {
        /* look in path for file - need to build the complete path
           skip if the complete path would be too long... */
        if (strlen(*path)+strlen(cmd)+2 <= MAX_PATH) {
            strncpy(buf,*path,MAX_PATH);
            strcat(buf,"/");
            strcat(buf,cmd);
            printf("checking %s\n",buf);

            /* not stat the complete path to see if the file exists */
            if (stat(buf,&sb)==0) {
                /* file exists - see if it is a regular, executable file*/
                /* this is really a bit more complex than shown here,
                   only checking for regular and executable by anyone */
                if ((S_ISREG(sb.st_mode)) && (sb.st_mode & S_IXOTH)) {
                    /* now fork, have the parent wait and tell the
                       child to exec the command */
                    if (fork()) {
                        /* parent just waits */
                        wait(&stat);
                        return(1);
                    } else {
                        /* Child execs */
                        execlp(cmd,cmd,NULL);
                        /* should never get here! */
                        exit(1);
                    }
                }
            }
            path++;
        }
    }
    return(0);
}

```

### Some Unix System Calls

```
pid_t fork(void);

int execl(const char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int setenv(const char *name, const char *value, int overwrite);

void unsetenv(const char *name);

int stat(const char *file_name, struct stat *buf);

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);

DIR *opendir(const char *name);

struct dirent *readdir(DIR *dir);
```

### One Win32 System Call

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPCTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

**Extra Credit (0 pts): How do you spell Windows?**

Question	Possible	Score
1	15	
2	25	
3	25	
4	16	
5	9	
6	10	
<b>Total</b>	<b>100</b>	

