# Exercise Class

Yi Shi
Fall 2018
Xi'an Jiaotong University

- **1.4** Which of the functionalities listed below need to be supported by the operating system for the following two settings: (a) handheld devices and (b) real-time systems.
  - a. Batch programming
  - b. Virtual memory
  - c. Time sharing

- **Answer:**
- For real-time systems, the operating system needs to support virtual memory and time sharing in a fair manner.
- For handheld systems, the operating system needs to provide virtual memory, but does not need to provide time-sharing.
- Batch programming is not necessary in both settings.

- **1.10** What is the purpose of interrupts? What are the differences between a trap and an interrupt? Can traps be generated intentionally by a user program? If so, for what purpose?

- **Answer:** An interrupt is a hardware-generated change-of-flow within the system. An interrupt handler is summoned to deal with the cause of the interrupt; control is then returned to the interrupted context and instruction. A trap is a software-generated interrupt. An interrupt can be used to signal the completion of an I/O to obviate the need for device polling. A trap can be used to call operating system routines or to catch arithmetic errors.

- **1.11** Direct memory access is used for high-speed I/O devices in order to avoid increasing the CPU´s execution load.
  - a. How does the CPU interface with the device to coordinate the transfer?
  - b. How does the CPU know when the memory operations are complete?
  - c. The CPU is allowed to execute other programs while the DMA controller is transferring data. Does this process interfere with the execution of the user programs? If so, describe what forms of interference are caused.

- **Answer:**
  - The CPU can initiate a DMA operation by writing values into special registers that can be independently accessed by the device. The device initiates the corresponding operation once it receives a command from the CPU.
  - When the device is finished with its operation, it interrupts the CPU to indicate the completion of the operation.
  - Both the device and the CPU can be accessing memory simultaneously. The memory controller provides access to the memory bus in a fair manner to these two entities. A CPU might therefore be unable to issue memory operations at peak speeds since it has to compete with the device in order to obtain access to the memory bus.

- **1.15** Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

- **Answer:** The processor could keep track of what locations are associated with each process and limit access to locations that are outside of a program's extent. Information regarding the extent of a program's memory could be maintained by using base and limits registers and by performing a check for every memory access.

- **2.2** List five services provided by an operating system that are designed to make it more convenient for users to use the computer system. In what cases it would be impossible for user-level programs to provide these services? Explain.

- **Answer:**
  - • **Program execution**. The operating system loads the contents (or sections) of a file into memory and begins its execution. A user-level program could not be trusted to properly allocate CPU time.
  - • **I/O operations**. Disks, tapes, serial lines, and other devices must be communicated with at a very low level. The user need only specify the device and the operation to perform on it, while the system converts that request into device- or controller-specific commands. User-level programs cannot be trusted to only access devices they should have access to and to only access them when they are otherwise unused.

- • **File-system manipulation**. There are many details in file creation, deletion, allocation, and naming that users should not have to perform. Blocks of disk space are used by files and must be tracked. Deleting a file requires removing the name file information and freeing the allocated blocks. Protections must also be checked to assure proper file access. User programs could neither ensure adherence to protection methods nor be trusted to allocate only free blocks and deallocate blocks on file deletion.
- • **Communications**. Message passing between systems requires messages be turned into packets of information, sent to the network controller, transmitted across a communications medium, and reassembled by the destination system. Packet ordering and data correction must take place. Again, user programs might not coordinate access to the network device, or they might receive packets destined for other processes.

– • **Error detection**. Error detection occurs at both the hardware and software levels. At the hardware level, all data transfers must be inspected to ensure that data have not been corrupted in transit. All data on media must be checked to be sure they have not changed since they were written to the media. At the software level, media must be checked for data consistency; for instance, do the number of allocated and unallocated blocks of storage match the total number on the device. There, errors are frequently process-independent (for instance, the corruption of data on a disk), so there must be a global program (the operating system) that handles all types of errors. Also, by having errors processed by the operating system, processes need not contain code to catch and correct all the errors possible on a system.

- **2.6** What are the advantages and disadvantages of using the same system call interface for manipulating both files and devices?

- **Answer**: Each device can be accessed as though it was a file in the file system. Since most of the kernel deals with devices through this file interface, it is relatively easy to add a new device driver by implementing the hardware-specific code to support this abstract file interface. Therefore, this benefits the development of both user program code, which can be written to access devices and files in the same manner, and device driver code, which can be written to support a well-defined API. The disadvantage with using the same interface is that it might be difficult to capture the functionality of certain devices within the context of the file access API, thereby either resulting in a loss of functionality or a loss of performance. Some of this could be overcome by the use of ioctl operation that provides a general purpose interface for processes to invoke operations on devices.

- **2.10** Why does Java provide the ability to call from a Java program native methods that are written in, say, C or C++? Provide an example of a situation in which a native method is useful.

- **Answer**: Java programs are intended to be platform I/O independent. Therefore, the language does not provide access to most specific system resources such as reading from I/O devices or ports. To perform a system I/O specific operation, you must write it in a language that supports such features (such as C or C++.) Keep in mind that a Java program that calls a native method written in an other language will no longer be architecture-neutral.

- **2.12** What is the main advantage of the microkernel approach to system design? How do user programs and system services interact in a microkernel architecture? What are the disadvantages of using the microkernel approach?

- **Anwser**: Benefits typically include the following
  - (a) adding a new service does not require modifying the kernel
  - (b) it is more secure as more operations are done in user mode than in kernel mode
  - (c) a simpler kernel design and functionality typically results in a more reliable operating system.
- User programs and system services interact in a microkernel architecture by using inter process communication mechanisms such as messaging. These messages are conveyed by the operating system.
- The primary disadvantage of the microkernel architecture are the overheads associated with inter process communication and the frequent use of the operating system's messaging functions in order to enable the user process and the system service to interact with each other.

- **2.14** What is the main advantage for an operating-system designer of using a virtual-machine architecture? What is the main advantage for a user?

- **Answer:** The system is easy to debug, and security problems are easy to solve. Virtual machines also provide a good platform for operating system research since many different operating systems may run on one physical system.

- **3.1** Describe the differences among short-term, medium-term, and long-term scheduling.

- **Answer:**
  - • **Short-term** (CPU scheduler)—selects from jobs in memory those jobs that are ready to execute and allocates the CPU to them.
  - • **Medium-term**—used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.
  - • **Long-term** (job scheduler)—determines which jobs are brought into memory for processing.
  - The primary difference is in the frequency of their execution. The short-term must select a new process quite often. Long-term is used much less often since it handles placing jobs in the system and may wait a while for a job to finish before it admits another one.

- **3.2** Describe the actions taken by a kernel to context-switch between processes.

- **Answer:** In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

- **3.4** Using the program shown in Figure 3.24, explain what will be output at Line A.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int value = 5;
int main()
{
  pid_t pid;
  pid_t=fork();
  if(pid ==0) {/*child process*/
  value += 15;
  }
else if (pid>0) {/* parent process */
  wait(NULL);
  printf("PARENT: value = %d", value); /* LINE A*/
  exit(0);
  }
}
```

- **Answer:** When control returns to the parent, its value remains at 5 as the child updates its copy of the value.

- **3.6** The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, .... Formally, it can be expressed as:

$$fib_0 = 0$$

$$fib_1 = 1$$

$$fib_n = fib_{n-1} + fib_{n-2}$$

- Write a C program using the fork() system call that that generates the Fibonacci sequence in the child process. The number of the sequence will be provided in the command line. For example, if 5 is provided, the first five numbers in the Fibonacci sequence will be output by the child process. Because the parent and child processes have their own copies of the data , it will be necessary for the child to output the sequence. Have the parent invoke the wait() call to wait for the child process to complete before exiting the program. Perform necessary error checking to ensure that a non-negative number is passed on the command line.

- **Answer:**

```
int main(int argc, char *argv[])
{
if (argc != 2 || atoi(argv[1]) < 0){
printf("Arguments Error!\n");
exit(0);}
//该程序用来检测输入是否正确，因为默认情况下，argv[0]中存放的是程序的名称，argv[1]中存放的是用户的第一个输入，以此类推。所以仅当用户输入一个数据时argc 的值为2并且输入的数非负时正确，否则检测到错误程序退出。
pid_t pid;      //在linux 下定义了一个进程
int i, a, b, fib;
int n = atoi(argv[1]);   //将字符串转化成整数类型
/* fork another process */
pid = fork();
if (pid < 0) { /* error occurred */
printf("Fork Failed\n");
exit(-1);
}
```

```
else if (pid == 0) { /* child process */
if (n == 0)
printf("0\n");
else if (n == 1)
printf("0, 1\n");
else if (n > 1) {
a = 0;
b = 1;
printf("0, 1");
for (i = 2; i <= n; i++) {
fib = a + b;
printf(",%d",fib);
a = b;
b = fib;
}
printf("\n");
}
}
else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
exit(0);
}
}
```

- **4.1** Provide two programming examples in which multithreading does *not* provide better performance than a single-threaded solution.

- **Answer:** (1) Any kind of sequential program is not a good candidate to be threaded. An example of this is a program that calculates an individual tax return. (2) Another example is a "shell" program such as the C-shell. Such a program must closely monitor its own working space such as open files, environment variables, and current working directory.

- **4.2** Describe the actions taken by a thread library to context switch between user-level threads.

- **Answer:** Context switching between user threads is quite similar to switching between kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its LWP and replacing it with another thread. This act typically involves saving and restoring the state of the registers.

- **4.3** Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

- **Answer:** When a kernel thread suffers a page fault, another kernel thread can be switched into use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multi-threaded solution would perform better even on a single-processor system.

- **4.4** Which of the following components of program state are shared across threads in a multithreaded process?
  - a. Register values
  - b. Heap memory
  - c. Global variables
  - d. Stack memory

- **Answer:** The threads of a multithreaded process share heap memory and global variables. Each thread has its separate set of register values and a separate stack.

- **4.8** Consider a multiprocessor system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios.
- a. The number of kernel threads allocated to the program is less than the number of processors.
- b. The number of kernel threads allocated to the program is equal to the number of processors.
- c. The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user-level threads.

- **Answer:**
  - When the number of kernel threads is less than the number of processors, then some of the processors would remain idle since the scheduler maps only kernel threads to processors and not user-level threads to processors.
  - When the number of kernel threads is exactly equal to the number of processors, then it is possible that all of the processors might be utilized simultaneously. However, when a kernel thread blocks inside the kernel (due to a page fault or while invoking system calls), the corresponding processor would remain idle.
  - When there are more kernel threads than processors, a blocked kernel thread could be swapped out in favor of another kernel thread that is ready to execute, thereby increasing the utilization of the multiprocessor system.

- **5.2** Discuss how the following pairs of scheduling criteria conflict in certain settings.
- a. CPU utilization and response time
- b. Average turnaround time and maximum waiting time
- c. I/O device utilization and CPU utilization

- **Answer:**
  - • CPU utilization and response time: CPU utilization is increased if the overheads associated with context switching is minimized. The context switching overheads could be lowered by performing context switches infrequently. This could however result in increasing the response time for processes.
  - • Average turnaround time and maximum waiting time: Average turnaround time is minimized by executing the shortest tasks first. Such a scheduling policy could however starve long-running tasks and thereby increase their waiting time.
  - • I/O device utilization and CPU utilization: CPU utilization is maximized by running long-running CPU-bound tasks without performing context switches. I/O device utilization is maximized by scheduling I/O-bound jobs as soon as they become ready to run, thereby incurring the overheads of context switches.

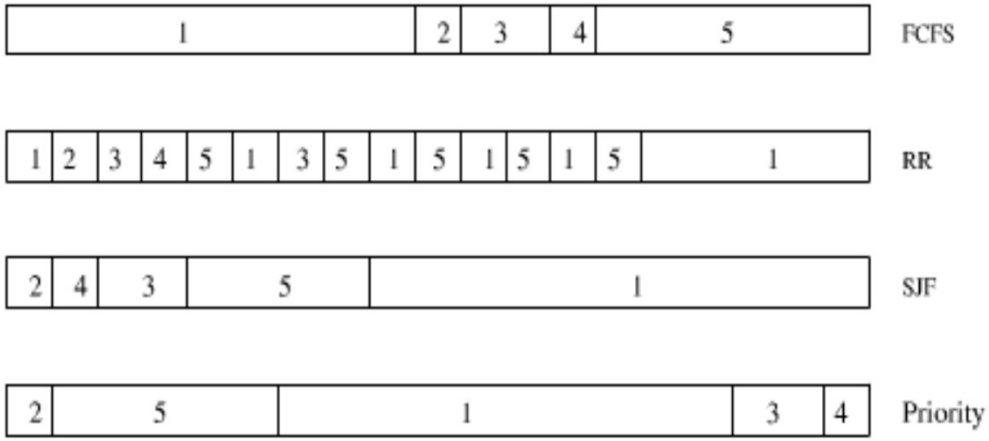
- **5.4** Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

| Process | Burst Time | Priority |
|---|---|---|
| *P*1 | 10 | 3 |
| *P*2 | 1 | 1 |
| *P*3 | 2 | 3 |
| *P*4 | 1 | 4 |
| *P*5 | 5 | 2 |

- The processes are assumed to have arrived in the order *P*1, *P*2, *P*3, *P*4, *P*5, all at time 0.

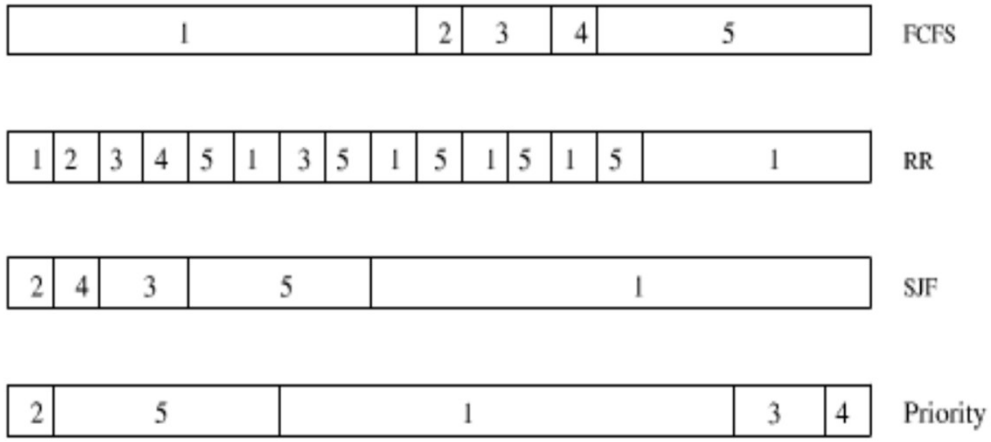
- a. Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF, a non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1) scheduling.
- b. What is the turnaround time of each process for each of the scheduling algorithms in part a?
- c. What is the waiting time of each process for each of the scheduling algorithms in part a?
- d. Which of the schedules in part results in the minimal average waiting time (over all processes)?

- **Answer:**
- The four Gantt charts are




- b. Turnaround time

| | FCFS | RR | SJF | Priority |
|---|---|---|---|---|
| *P*1 | 10 | 19 | 19 | 16 |
| *P*2 | 11 | 2 | 1 | 1 |
| *P*3 | 13 | 7 | 4 | 18 |
| *P*4 | 14 | 4 | 2 | 19 |
| *P*5 | 19 | 14 | 9 | 6 |




- c. Waiting time (turnaround time minus burst time)

| | FCFS | RR | SJF | Priority |
|---|---|---|---|---|
| *P*1 | 0 | 9 | 9 | 6 |
| *P*2 | 10 | 1 | 0 | 0 |
| *P*3 | 11 | 5 | 2 | 16 |
| *P*4 | 13 | 3 | 1 | 18 |
| *P*5 | 14 | 9 | 4 | 1 |

- d. Shortest Job First

- **5.5** Which of the following scheduling algorithms could result in starvation?
  - a. First-come, first-served
  - b. Shortest job first
  - c. Round robin
  - d. Priority

- **Answer**: Shortest job first and priority-based scheduling algorithms could result in starvation.

- **6.2** The first known correct software solution to the critical-section problem for *n* processes with a lower bound on waiting of *n* − 1 turns was presented by Eisenberg andMcGuire. The processes share the following variables:

  enum pstate *{*idle, want in, in cs*}*;

  pstate flag[n];

  int turn;

- All the elements of flag are initially idle; the initial value of turn is immaterial (between 0 and n-1). The structure of process *Pi* is shown in Figure 6.26. Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do{
    while(TRUE){
        flag[i]=want_in;
        j=turn;
        while(j!=i){
            if(flag[j]!=idle)
                j=turn;
            else
                j=(j+1)%n;
            }
        flag[i]=in.cs;
        j=0;
        while((j<n)&&(j==i||flag[j]!=in.cs))
            j++;
        if((j>=n)&&(turn==i||flag[turn]==idle))
            break;
    }
// critical section
j=(turn+1)%n;
while(flag[j]==idle)
    j=(j+1)%n;
turn=j;
flag[i]=idle;
//remainder section
}while(TRUE);
```

- **Answer:** This algorithm satisfies the three conditions. Before we show that the three conditions are satisfied, we give a brief explanation of what the algorithm does to ensure mutual exclusion. When a process *i* requires access to critical section, it first sets its flag variable to want_in to indicate its desire. It then performs the following steps: (1) It ensures that all processes whose index lies between *turn* and *i* are idle. (2) If so, it updates its flag to in_cs and checks whether there is already some other process that has updated its flag to in_cs. (3) If not and if it is this process's turn to enter the critical section or if the process indicated by the turn variable is idle, it enters the critical section. Given the above description, we can reason about how the algorithm satisfies the requirements in the following manner:

- Mutual exclusion is ensured: Notice that a process enters the critical section only if the following requirements are satisfied: no other process has its flag variable set to in_cs. Since the process sets its own flag variable set to in_cs before checking the status of other processes, we are guaranteed that no two processes will enter the critical section simultaneously.

- Progress requirement is satisfied: Consider the situation where multiple processes simultaneously set their flag variables to in_cs and then check whether there is any other process has the flag variable set to in_cs. When this happens, all processes realize that there are competing processes, enter the next iteration of the outer *while*(1) loop and reset their flag variables to want_in. Now the only process that will set its turn variable to in_cs is the process whose index is closest to turn. It is however possible that new processes whose index values are even closer to turn might decide to enter the critical section at this point and therefore might be able to simultaneously set its flag to in_cs. These processes would then realize there are competing processes and might restart the process of entering the critical section. However, at each iteration, the index values of processes that set their flag variables to in_cs become closer to turn and eventually we reach the following condition: only one process (say k) sets its flag to in_cs and no other process whose index lies between turn and *k* has set its flag to in_cs. This process then gets to enter the critical section.

- Bounded-waiting requirement is met: The bounded waiting requirement is satisfied by the fact that when a process *k* desires to enter the critical section, its flag is no longer set to idle. Therefore, any process whose index does not lie between turn and *k* cannot enter the critical section. In the meantime, all processes whose index falls between turn and *k* and desire to enter the critical section would indeed enter the critical section (due to the fact that the system always makes progress) and the turn value monotonically becomes closer to *k*. Eventually, either turn becomes *k* or there are no processes whose index values lie between turn and *k*, and therefore process *k* gets to enter the critical section.

- **6.11 The Sleeping-Barber Problem**. A barbershop consists of a waiting room with *n* chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

- **Answer:**

```
int count=0;
semaphore mutex={1},sofa={N},empty={1},full={0},cut={0},payment={0},receipt={0};
guest()
{wait(mutex);
if(count>N) {signal(mutex); exit shop;}
else {count:=count+1;
        if (count>1) {signal(mutex); wait(sofa); sit on sofa; wait(empty); get up from sofa;
                signal(sofa);}
        else {signal(mutex); wait(empty); sit on the baber_chair; signal(full); wait(cut); pay;
                signal(payment); wait(receipt); get up from the baber_chair; signal(empty);
                wait(mutex); count:=count-1; signal(mutex); exit shop;}
}
}
barber()
{ while(1)
{wait(full); cut chair; signal(cut); wait(payment); accept payment; signal(receipt); }
}
```