Chapters1and2

February 24, 2023 11:17 AM

Review Questions – Chapters 1 and 2

Operating Systems SFWRENG 3SH3 Term 2, Winter 2023
Prof. Neerja Mhaskar

1 Distinguish between system and application programs.

2 Explain the purpose and significance of an interrupt vector.

3 What is a bootstrap program, and where is it stored?

4 What role do device controllers and device drivers play in a computer system?

5 What are the advantages of using a multi-core system?

6 Why are at least two modes (kernel and user mode) necessary in a computer system?

7 What are the advantages and disadvantages of the microkernel approach to structure an Operating System?

8 What are the advantages and disadvantages of the Modular approach to structure an Operating System? Why should every operating system support modules?

9 What are System Calls and what are they used for? Is it true that all system calls are programmed in high level languages? Justify your answer.

10 Why do you need an API to access system calls? What are the advantages of using the System call Interface?

11 What are system daemons? What are the advantages of using system programs. Why not include their functionality in the operating system itself?

1. System programs serve to help the OS run, application programs are user programs like MS word
2. An interrupt vector is a vector containing the addresses of the ISR corresponding to each type of interrupt, so that the OS knows what to do when a certain interrupt occurs
3. A bootstrap program is a program that starts the operating system, it is stored in the EEPROM
4. Device controllers: part of the device, facilitates communication between device and driver
   Device drivers: one per controller, communicate between device controller and OS
5. Multi-core systems allow for parallel computing, which means many things can happen concurrently, which makes operation faster & more responsive (throughput, economy of scale, reliability)
6. Kernel mode can make system calls, user mode is a layer of protection that sits as a barrier between the user and kernel-level operations, s.t. the user cannot change characteristics of the OS
7. Microkernel: remove anything except the absolute essentials from the kernel.
   a. pros: fast and safe, easy to port to new architectures
   b. cons: communication overhead
8. Modular: kernel is made up of separate core components, anything else is added on as a module
   a. pros: modularity, flexibility - everything is loaded as needed
   b. cons: complexity and therefore reliability
   c. every operating system should support modules since it allows for high flexibility and additional features to be added easily as modules. Makes changes to the platform easy.
9. System calls are functions available to the kernel that perform kernel-level operations, like fork() and wait()
   a. provides an interface to OS services
   b. available to user via API like win32
   c. programmed in high level languages because they need to be fast and precise
10. You need an API to access system calls for safety reasons
    a. user mode can't access sys calls, so use API to get kernel mode to do it for you
       i. the bells and whistles are hidden from the user = main advantage
11. System daemons are system processes that are constantly running in the background. They are essential in supporting the OS's operation.
       i. ex: program execution support, program loading
    a. system programs are programs that are associate with the OS but not necessarily the kernel
       i. ex: UI for system call

a. system programs are programs that are associate with the OS but not necessarily the kernel
    i. ex: UI for system call

Chapters3and4

February 24, 2023 11:18 AM

Review Questions – Processes & Threads and Concurrency (Chapters 3 and 4)

Operating Systems SFWRENG 3SH3 Term 2, Winter 2023

Prof. Neerja Mhaskar

Q1) Using the program below explain what the output will be at LINE A and why?

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
pid_t pid;

  pid = fork();

  if (pid == 0) { /* child process */
    value += 15;
    return 0;
  }
  else if (pid > 0) { /* parent process */
    wait(NULL);
    printf("PARENT: value = %d",value); /* LINE A */
    return 0;
  }
}
```

Q2) Including the initial parent process, how many processes are created by the program shown below? Construct a tree of processes as explained in class for the processes created.

1) The output would be 5, since value is only incremented inside the child process

```
#include <stdio.h>
#include <unistd.h>

{
int main()
{
  int i;

{
  for (i = 0; i < 4; i++)
   fork();
{

  return 0;
}
```

Q3) Using the program below, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

2) 2^4 = 16

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

  /* fork a child process */
  pid = fork();

  if (pid lt; 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
  }
  else if (pid == 0) { /* child process */
    pid1 = getpid();
    printf("child: pid = %d",pid); /* A */
    printf("child: pid1 = %d",pid1); /* B */
  }
  else { /* parent process */
    pid1 = getpid();
    printf("parent: pid = %d",pid); /* C */
    printf("parent: pid1 = %d",pid1); /* D */
    wait(NULL);
  }
  return 0;
}
```

Q4) When a process creates a new process using the `fork()` operation, which of the following states is shared between the parent process and the child process?
a. Stack
b. Heap
c. Shared memory segments

Q5) Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

Q6) Distinguish between parallelism and concurrency.

Q7) Distinguish between data and task parallelism.

3)
**getpid() gets the current process's PID as seen by the OS
**pid is the return value of the fork() sys call, which is different for the parent and child

A: 0
B: 2603
C: 2600
D: 2603

4) Shared memory

5) S = 0.2
gain = 1/(S + (1-S)/N)
   a. g = 1/(0.4 + (0.6)/2) = 1.4286
   b. g = 1/(0.4 + (0.6/4)) = 1.8182

6) Parallel - more than 1 task at a time
Concurrent - more than 1 task making progress at a time

7) Data parallelism - on each core, perform the same operation on subsets of data
Task parallelism - on each core, perform different tasks on the same data
   - threading across cores

Q8) For the program below:

```
int main() {
      Pid_t pid1, pid2;
      Pid1 = fork();
      pid2 = fork();
      if (pid1 == 0) { /* child process */
      pthread_create(. . .);
      }
      if (pid2 == 0) { /* child process */
      pthread_create(. . .);
      }
}
```

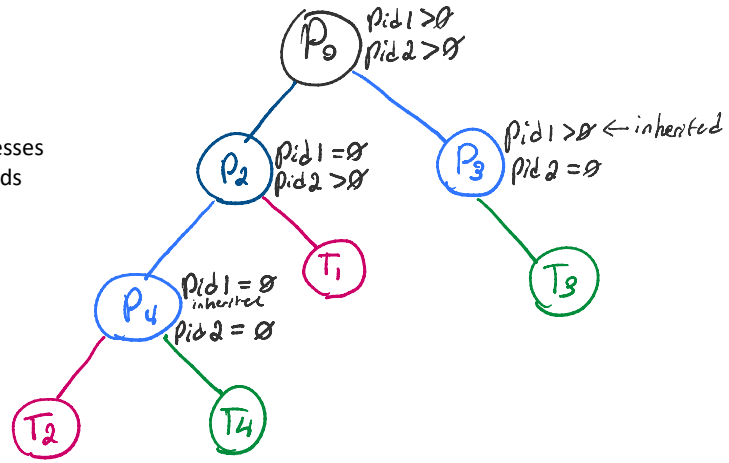
a. How many unique processes are created?
b. How many unique threads are created?
c. Draw the process and thread tree for this code.

Q9) The program shown in Figure 4.16 (page 194 of the textbook) uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

 pid = fork();

 if (pid == 0) { /* child process */
   pthread_attr_init(&attr);
   pthread create(&tid,&attr,runner,NULL);
   pthread_join(tid,NULL);
   printf("CHILD: value = %d",value); /* LINE C */
 }
 else if (pid gt; 0) { /* parent process */
   wait(NULL);
   printf("PARENT: value = %d",value); /* LINE P */
 }
```

8)

a. 4 processes
b. 4 threads



9) C: 5, P: 0

```
void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```

Q10) Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores 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 processing cores.
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

Q 11) A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up when a single file must be opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

1. How many threads will you create to perform the input and output? Explain.
2. How many threads will you create for the CPU-intensive portion of the application? Explain.

10

a) suboptimal, we want to maximize the use of resources so we should allocate more kernel threads to the program in order to increase throughput
b) optimal, assigning one kernel thread to each processing core maximizes the use of resources
c) has consequences: context switching needs to be performed more often, comes with a lot of overhead

11

a) currently the application is single threaded, and it should remain that way for the input and output so to avoid multiple threads modifying shared resources.
   Since only one file is being read from and written to, we do not need to multithread this task and use synchronization tools to ensure that the shared resources do not get modified incorrectly.
   a. it is simpler and easier if we use one thread for input and one for output
   b. reading/writing is a blocking operation!!!!
b) Since the system has four processors available, we should aim to have 4 threads (user/kernel) in order to maximize resource usage.

Chapter 6 and 7

February 24, 2023 11:17 AM

Review Questions – Synchronization Tools and Examples (Chapters 6 and 7)

Operating Systems SFWRENG 3SH3 Term 2, Winter 2023
Prof. Neerja Mhaskar

1. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the amount that is to be deposited or withdrawn from the bank account. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function and the wife calls `deposit()`. Describe with an example how a race condition is possible and what might be done to prevent the race condition from occurring.

2. Prove that the following critical solution problem fails. Also state explicitly state which of the critical section problem solution requirements are not satisfied and why.
Note: We assume that load and store machine language instructions are atomic.

   The two processes share the following variable:
   `int turn;`
   a. **`turn`** indicates whose turn it is to enter the critical section.
   b. You can initialize `turn = 0` or `turn = 1`. The failure of the solution is independent of the initialization.

   Algorithm for process P_i

```
do {
     while (turn==j);
     critical section
     turn = j;
     remainder section
} while (true);
```

3. Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:
   a. The lock is to be held for a short duration.
   b. The lock is to be held for a long duration.
   c. A thread may be put to sleep while holding the lock.

4. Illustrate how you would use semaphores to synchronize processes P1, P2, P3, P4 where P1 is executed before P2, P2 is executed before P3, P3 is executed before P4. Write pseudocode that Illustrates how a binary semaphore can be used to implement mutual exclusion among *n* processes.

5. Consider two concurrently running processes P_1 and P_2 that require P_1 to execute before

1. husband calls withdraw, wife calls deposit. In the instance where a race condition occurs and the withdrawal is interrupted to do the deposit.

balance is $200;
husband calls withdraw $150;
shared balance local to husband = $50;
before withdraw is committed, wife calls deposit $50;
shared balance is now $100, but husband sees it as $50;

to prevent race condition, use mutex lock to make withdraw and deposit processes atomic.

2.
conditions: mutual exclusion, bounded waiting, progression;
- mutual exclusion: yes, spinlock ensures mutex
- progression: yes, implied critical section can only run once before giving up turn
- bounded waiting: no, if P_1 gets stuck in CS, P_2 will wait indefinitely for P_1 to finish and change turn

3.
a) spinlock: less overhead than mutex lock, minimal waste of resources because very short hold
b) mutex lock: spinlock being held for a long time would waste a lot of resources, use mutex lock instead
c) mutex lock: lock may be held indefinitely, use mutex to avoid wasting resources if lock is held for a long time

4.

```
sem_t S1, S2, S3;
Process 1{
crit section 1
signal S1
}
```

```
Process 2{
wait S1
CS
signal S2
```

```
Process 3{
wait S2
CS
signal S3
```

```
Process 4{
wait S3
CS
}
```

pseudocode that Illustrates how a binary semaphore can be used to implement mutual exclusion among *n* processes.

5. Consider two concurrently running processes P_1 and P_2 that require P_1 to execute before P_2 (here P_1 and P_2 may or may not access/modify shared variables.). Use semaphores to synchronize processes P_1 and P_2.

```
Process 1{
crit section 1
signal S1
}

Process 2{
wait S1
CS
signal S2
}

Process 3{
wait S2
CS
signal S3
}

Process 4{
wait S3
CS
}
```

5. sem S

```
P1{
crit section
signal S
}

P2{
wait(S)
crit section
signal(S)
}
```

6. Illustrate how you would use semaphores to synchronize processes P_1, P_2, P_3, P_4 where P_1 is executed before P_2, P_2 is executed before P_3, P_3 is executed before P_4.

7. What is the difference between a semaphore and conditional variable?

8. Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

9. In the solution for dining philosophers' problem using monitors, provide a scenario in which a philosopher may starve to death.

10. Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

11. Implement a mutex lock using the `test_and_set()` atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
bool held;
} lock;
```

`held == false (true)` indicates that the lock is available (not available) Using the `struct lock`, illustrate how the following functions can be implemented using the `test_and_set()` instructions:

a. `void acquire(lock *mutex)`
b. `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

7. Semaphore waits until it can be decremented and then allows code to execute, conditional var executes code in an if statement based on assignments. Conditionals make a program wait until a condition is met before executing, semaphores limit the number of threads on a shared resource.

8.

```
monitor{
    sem_t full = 0, empty = N;
    add{
        wait(empty);
        //add to buffer
        signal(full);
    }
    subtract{
        wait(full);
        //take from buffer
        signal(empty);
    }
}
```

9. a pilosopher may starve to death if one of the philosophers dining next to them never gives up their shared chopstick, causing them to circularly wait on one another to be done with their chopsticks i.e. if a deadlock occurs.

10. Interrupts can only stop the execution of a single core. If an interrupt is called, the isr routine is run, but the other cores continue to execute their commands and access shared resources, leading to inconsistency in the data.

11.

```
test_and_set(x) --> temp= x,  x = true, return temp;
void release(lock *mutex){
    mutex->held = false;
}
void acquire(lock *mutex){
    while(test_and_set(mutex->held)){spin};
}
```

Chapter 8

February 24, 2023 11:17 AM

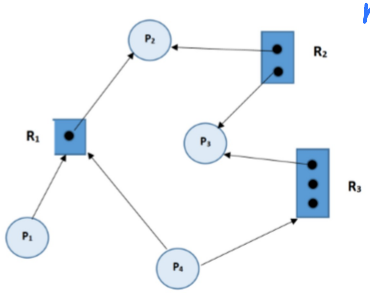
Review Questions – Deadlocks (Chapter 8)

Operating Systems SFWRENG 3SH3 Term 2, Winter 2023

Prof. Neerja Mhaskar

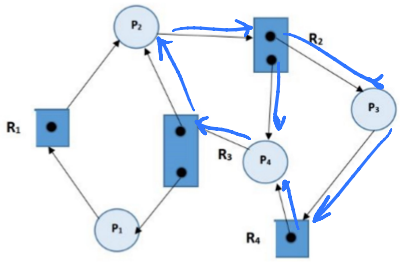
Questions:

1. Consider the below resource allocation graph. Is the system in a deadlock state? If so, report the cycle(s) causing deadlock. If not, explain the order in which processes access the resources requested and complete execution.



no deadlock
P_2 holds R_1 & R_2, P_3 holds R_2 & R_3
P_4 is granted R_3
P_2 finishes execution
P_1 is granted R_1
P_1 finishes execution
P_4 is granted R_1
P_4 finishes execution

2. Consider the below resource allocation graph. Is the system in a deadlocked state? If so, report the cycle(s) causing deadlock. If not, explain the order in which processes access the resources requested and complete execution.



Deadlock
$P_3 \rightarrow R_4 \rightarrow P_4 \rightarrow R_3 \rightarrow P_2 \rightarrow R_2 \rightarrow P_3$

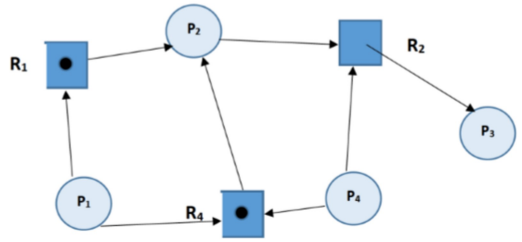
3. Consider the following snapshot of a system:

	Allocation	Max	Available	Need
	A B C D	A B C D	A B C D	A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0	0 0 0 0
P_1	1 0 0 0	1 7 5 0		0 7 5 0
P_2	1 3 5 4	2 3 5 6		1 0 0 2
P_3	0 6 3 2	0 6 5 2		0 0 2 0
P_4	0 0 1 4	0 6 5 6		0 6 4 2

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix Need?
b. Is the system in a safe state?
c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

4. Consider the below resource allocation graph. Construct the corresponding wait-for graph. Is the system in deadlock? If so, provide the cycle causing deadlock.



5. Consider the following snapshot of a system at time *T*0:

Five processes ***P*0** through ***P*4**.

Three resource types A (10 instances), *B* (3 instances), and *C* (6 instances)

Snapshot at time ***T*0**:

	Allocation	Max	Available	Need
	A B C D	A B C D	A B C D	A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0	0 0 0 0
P_1	1 0 0 0	1 7 5 0		0 7 5 0
P_2	1 3 5 4	2 3 5 6		1 0 0 2
P_3	0 6 3 2	0 6 5 2		0 0 2 0
P_4	0 0 1 4	0 6 5 6		0 6 4 2

Three resource types A (10 instances), *B* (3 instances), and *C* (6 instances)

Snapshot at time T_0:

	Allocation A B C	*Request* A B C	*Available* A B C
P0	2 1 1	0 0 0	0 0 0
P1	2 1 2	2 0 2	
P2	4 0 0	0 0 1	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

W = 000
P_0 has $R \leq W$
$W = W + A = 211$
P_2 has $R \leq W$
$W = 211 + 400$
$= 611$
P_3 has $R \leq W$
$W = 611 + 211$
$= 822$

$P_1 \Rightarrow R \leq W$
$W = 822 + 212$
$= 1034$
$P_4 \Rightarrow R \leq W$
$W = 1036$

a) Is the system in deadlocked state? If no, provide a sequence of processes satisfying the safety requirement. If yes, explain why and list the processes involved in the deadlock. No deadlock: $P_0 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1 \rightarrow P_4$

b) Suppose process P1 makes an additional request of resource type B, the Request matrix is modified as follows:

	Request A B C
P_0	0 0 0
P_1	2 1 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

c) Is the system in deadlocked state? If no, provide a sequence of processes satisfying the safety requirement. If yes, explain why and list the processes involved in the deadlock.

I'm pretty sure its fine

Tutorial

February 28, 2023 2:49 PM

- its impossible to have multiple kernel threads to a user thread because the OS doesn't know what resources to allocate

Many to many

- if kernel threads<cores
    - some cores remain idle, user-threads can't be directly mapped to cores
- kernel threads == cores
    - optimal use of cores
    - when a kernel threads blocks (i.e. sys call like wait or join which would cause the thread to wait on another thread), then processor remains idle
- kernel threads > cores (for many to many)
    - increased utilization
    - optimal use + kernel threads are swapped out when one of them blocks s.t. core doesn't have to be idle

Practice lab 4 notes

- banking system
    - can deposit or withdraw, that's it
- concept
    - let there be a bank account limit of $400
    - let the deposit/withdrawals only be done in increments of $100
    - therefore we can make maximum 4 deposits and maximum 4 withdrawals
    - use semaphores to keep track of how many deposits or withdrawals can be mae
        - two counting semaphores
        - sem_wait() - waits for S > 0 then decrements
        - sem_post() - waits for S < MAX then increments
    - init… semaphore dep_S = deposit(0, 4, 4); sempahotre with_S = withdrawal(0, 4, 0);
- A husband and wife want to make a deposit/withdrawal at the same time. What can be done to prevent a race condition?
        - both threads want to access and change shared data at the same time
            - critical section - section of code that needs to be run atomically
                - atomic code - operation that is guaranteed to be isolated from other operations happening at the same time (indivisible)
    - the two transactions will be serialized, starting with $250, the husband withdraws 50, so the local value of the balance for him becomes $200
    - before he can commit the transaction, the wife deposits $100 and updates the shared balance to $350
    - so the husband sees the shared balance as $200 (not right)

Critical section requirements

- Mutual exclusion
    - no other process is allowed to execute inside the crit section

- Progress
    - processes shouldn't have to wait indefinitely to enter crit section
- Bounded waiting
    - a bound on the number of times a process is allowed to execute the critical section

Question 6

Prove that the following critical solution problem fails. Also explicitly state which of the critical section problem solution requirements are not satisfied and why.

- assume load and store machine instructions are atomic
    - load: move from mem to reg
    - store: move from reg to mem
- the two processes share the following var int turn
    - turn indicatesd whose turn it is to enter the critical section
    - you can initialize turn = 0 or 1. The failure of the solutions I sindepending of the initialization

Algo for process P_0	Algo for process P_1

do{
while(turn == 0){ //spin
//crit section
turn = 1; //switch
}
//remainder section
}while(true);
	do{
	while(turn == 1){ //spin
	//crit section
	turn = 0 //switch
	}
	//remainder section
	}while(true);

Solution

- Mutual exclusion?
    - while(turn) ensures mutual exclusion, we good
- Progress
    - turn switching isnt enough: if p0 gets stuck waiting, then p1 has to wait indefinitely for p0 to exit the while loop
    - p1 can't start until p0 switches the turn flag in the exit section
- Bounded waiting
    - bound on number of times process can execute CS
    - there's a theoretical bound of one, as the switch flag is changed at every critical section's exit -- satisfied
        - can't run the critical section more than once before giving up its turn: **bound on the number of times the critical section can run**
        - a process is always selected to execute after P_i

Spin and mutex

Assume a system has multiple cores. For each scenario, which is a better locking mechanism: spinlock or mutex lock. What if a waiting processes sleep while waiting holding lock?

- causes thread to wait in a loop while repeatedly checking if lock is avaiable

    - while loop spinning
- mutex lock
    - binary var who provides locking mechanism
    - mutex lock behaves as an actual lock, semaphore only behaves as a var counter that checks for a condition

1. Lock is held for a short duration
    a. spinlock - only happens for a brief moment
2. Lock is held for a long duration
    a. mutex lock with waiting
3. Thread put to sleep while holding lock
    a. mutex lock: we don't know how much time its gonna take for the thread to wake up, so don't use a spin, it might waste resources indefinitely

Semaphores
Illustrate how you would use semaphores to synchronize 4 processes to ensure that p1 runs before p2 runs before p3 runs before p4