CS 594: Operating Systems, Spring 2008

Dr. Beck

Final Exam, 1 May 2008

1. (20 points)

Consider two processes in a system where allocation of a process that is already allocated will block until the resource is freed:

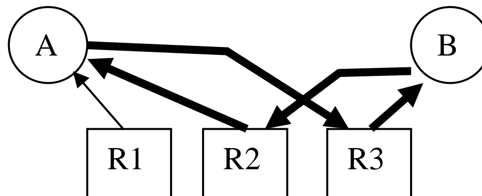
Process A	Process B
`Allocate R1`	`Allocate R3`
`Allocate R2`	`Allocate R2`
`Allocate R3`	`Allocate R1`
`do_A(R1,R2,R3)`	`do_B(R1,R2,R3)`
`Free R1`	`Free R3`
`Free R2`	`Free R2`
`Free R3`	`Free R1`

a) Demonstrate that deadlock is possible in this system by showing a sequence of operations that does deadlock. Illustrate by showing a resource allocation graph for the deadlocked state.

Answer:

Process A	*Process B*
Allocate R1	-
-	*Allocate R3*
Allocate R2	-
-	*Allocate R2*
Allocate R3	-

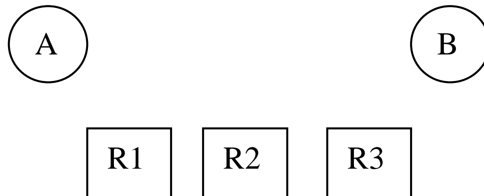
-- deadlocked --



b) Demonstrate that deadlock is not inevitable in this system by showing a sequence of operations that does not deadlock. Illustrate the state of the system when Process A completes execution using a resource allocation graph.

Answer:

Process A	*Process B*
`Allocate R1`	-
`Allocate R2`	-
`Allocate R3`	-
`do_A(R1,R2,R3)`	-
`Free R1`	-
`Free R2`	-
`Free R3`	-
-	`Allocate R3`
-	`Allocate R2`
-	`Allocate R1`
-	`do_B(R1,R2,R3)`
-	`Free R3`
-	`Free R2`
-	`Free R1`



c) Explain a deadlock-avoidance mechanism that could be used in this system and which would ensure that deadlock does not occur. How do you know that it would avoid deadlock in all scenarios (not just this one)?

Answer: Requiring that all allocations be made in a single atomic action would ensure that the system was deadlock-free by eliminating the possibility of hold-and-wait, which involves a process allocating some resources an then blocking on others. Without hold-and-wait, deadlock cannot occur.

2. (15 points) Explain how the convoy problem can occur in a process scheduling system with a single queue. Explain how the use of a Multilevel Feedback Queue can avoid the convoy problem.

Answer: The convoy problem occurs when a single large CPU-bound job shares the ready-to-run queue with a large number of IO-bound jobs in a non-preemptive scheduling system. The result is that the IO jobs spend a large amount of time waiting for the CPU job to relinquish the processor, at which point they can all run quickly, initiating IO operations and then relinquishing the CPU and, IO utilization is low.

3. (15 points) Explain how an inverted page table is implemented. What is the advantage of using an inverted page table? Why is the Translation Lookaside Buffer especially important when using an inverted page table?

Answer: An inverted page table has one entry for every physical memory frame in the system, with the entry for physical frame stored a position i within the table. Allocated frame i has the page index used in virtual addresses stored in the inverted table at position i. Given a virtual address (pageno, offset), the inverted table is scanned from the beginning in a linear fashion to find the entry storing pageno. If none is found, there is an addressing error.

The TLB is especially important when using an inverted page table because of the high cost of searching the table linearly. When a page resides within the TLB, the linear search can be abandoned when the hit is discovered. Only TLB misses must complete the linear search, and addressing errors must still search the entire table.

4. (15 points) Explain how copy-on-write can be used to optimize the fork-exec sequence of system calls in Unix/Linux. Why is it especially effective for this sequence of operations?

Answer: Copy-on-write takes advantage of the fact that most copied pages are used in an unmodified form, with neither the original or the copy being modified during the lifetime of the copy. It works by copying the page table entries associated with the data to be copied rather than the data itself. Both the original and the copied entries are write-protected so that no modifications can be made, but a page protection exception occurs if an attempt is made to modify either. At that point the data itself is copied (hence the name copy-on-write) and the modifying operation is restarted.

In the fork-exec sequence, the copied data segment and most of the stack segment from the original process are never modified, but the copy is instead overwritten by the exec system call. Thus, copy-on-write can allow the child process to be created by copying the parent's page table, and to run a very short time, only long enough to invoke the exec system call, modifying only minimal portions of the parent's data. It is especially effective because so little of the parent's data segment is typically modified before the exec call in made.

5. (20 points) Consider this code taken from the implementation of kthreads:

```
id KtSched()
{
        K_t kt;
        Jval j_kt;
        JRB jb;
        unsigned int sp;
        unsigned int now;
        Dllist dtmp;
        JRB tmp;

        if(setjmp(ktRunning->jmpbuf) != 0)
        {
                FreeFinishedThreads();
                if(ktRunning->die_now) kt_exit();

                return;
        }
start:

        if (!jrb_empty(ktSleeping)) {
          now = time(0);
          while(!jrb_empty(ktSleeping))
          {
                kt = (K_t) jval_v(jrb_val(jrb_first(ktSleeping)));
                if(kt->wake_time > now) break;
                WakeKThread(kt);
          }
        }
        …
```

Explain the purpose of the call to `setjmp` in this instance. Explain what the scheduler does upon the first return from `setjmp` and on the second return. Include the functioning of the conditional that occurs immediately after the label `start` including descriptions of the data structure accessed and the behavior of any functions called.

Answer:The call to setjmp serves to save the state of running thread to the jmpbuf data structure, allowing it to be later used in a longjmp call. The first return from setjmp will occur after the state has been saved, and will return zero, at which point the scheduler will continue executing at the lable "start", searching the ktSleeping jrb for sleeping jobs that need to be woken up (because their wake_time is not greater than now). The routine WakeKThread is calle on any job that needs to be awakened.

The second return from setjmp will occur when longjmp is called on the jmpbuf data structure that was saved during the call to setjmp. This return is identical to the first, except for the fact that its return value is non-zero. This is detected by the conditional, which frees threads and then if the thread that has been invoked has flagged that it should die, the scheduler will exit. If there is no such flag, the scheduler proceeds to process the ktSleeping jrb as above.

6. (15 points) Explain *in detail* exactly how a malignant user may take advantage of the following program to breach security. In your explanation, give what conditions must be present (i.e. who owns the program, who is running the program, what kind of security breach can occur and exactly how the breach occurs).

Assume the network connection is a file descriptor for a standard socket connection.

```
char *get_text(int fd)
{
  char buffer[100];
  int i;
  int c;

  i = 0;
  while(1) {
    n = read(fd, &c, 1);
    if (n == 0) {
      buffer[i] = '\0';
      return strdup(buffer);
    } else {
      buffer[i] = c;
      i++;
    }
  }
}
```

```
main()
{
  int fd;
  char *lines[10000];
  int nl;
  int beginning;

  for (i = 0; i < 10000; i++) {
    fd =
       get_network_connection();
    lines[i] = get_text(fd);
  }
  for (i = 9999; i >= 0; i--)
printf("%d %s", i, lines[i]);
```

Answer: The contents of the program stack are as follows:

```
sp              int c;
sp+4            int i;
sp+8            char buffer[100];
sp+108          <subroutine linkage: sp, pc>
sp+116          int beginning;
sp+120          int nl;
sp+124          char *lines[10000];
sp+10124        int fd;
```

A buffer overflow attack can be created by writing more than 100 bytes of input, overflowing the buffer at sp+8 and writing onto the lower portion of the program stack. The first area to be overwritten is the subroutine linkage, which includes the saved pc which determines the return location when the subroutine returns. This can be overwritten with location sp+116, and the malicious code can then be written starting at that location. When the program get_text returns, it will execute the code at sp+116 instead of the calling code, and the process will then be under the control of the attacker. However the permissions and identity of the process will still be that of whoever originally invoked the main program. If the program is running with high permissions such as root, this constitutes a very serious breach.