COMP3231/9201/3891/9283 Operating Systems 2020/T1

Tutorial Week 3

Questions and Answers

System Call Interface

1. The following segment of code is similar (but much simpler) to the main task that the daemon inetd performs. It accepts connections on a socket and forks a process to handle the connection.

   This is not guaranteed to be compilable. Use the man command if you want to investigate what all the system calls are doing.

```
0001 xxx(int socket){
0002
0003         while ((fd = accept(socket, NULL, NULL)) >= 0) {
0004                 switch((pid = fork())) {
0005                 case -1:
0006                         syslog(LOG_WARN, "%s cannot create process: %s",
0007                             progname, sys_error(errno));
0008                         continue;
0009                 case 0:
0010                         close(0);
0011                         close(1);
0012                         dup(fd);
0013                         dup(fd);
0014                         execl("/usr/sbin/handle_connection",
0015                             "handle_connection", NULL);
0016                         syslog(LOG_WARN, "%s cannot exec handle_connection\
0017                             helper : %s", progname, sys_error(errno));
0018                         _exit(0);
0019                 default:
0020                         waitpid(pid, &status, 0);
0021                         if (WIFEXITED(status) && WIFEXITSTATUS(status) == 0)
0022                                 continue;
0023                         syslog(LOG_WARN, "handle_connection failed:\
0024                             exit status +%d\n", status);
0025                 }
0026         }
0027 }
```

   a. Identify which lines of code are executed by the parent process.
   b. Identify which lines of code are invoked by the child process.
   c. Under what circumstances does the child terminate?

      a. The parent process can execute 0001 - 0004 (fork()), 0005 - 0008 (if the fork() fails) whereupon it continues to wait for a connection on the socket, and 0019 - 0026 if the fork() succeeds. By calling waitpid(), the parent suspends execution until the child terminates. It exits the while loop when the accept() fails (see "man exec" for failure modes).

b. The child returns at line 004, and then executes 0009 - 0015, only executing 0016 - 0018 if the execl() fails.
c. The child terminates when the program /usr/sbin/handle_connection terminates (assuming the exec was successful), but otherwise if the exec fails.

Concurrency and Deadlock

2. For each of the following scenarios, one or more dining philosophers are going hungry. What is the condition the philosophers are suffering from?

   a. Each philosopher at the table has picked up his left fork, and is waiting for his right fork
   b. Only one philosopher is allowed to eat at a time. When more than one philosophy is hungry, the youngest one goes first. The oldest philosopher never gets to eat.
   c. Each philosopher, after picking up his left fork, puts it back down if he can't immediately pick up the right fork to give others a chance to eat. No philosopher is managing to eat despite lots of left fork activity.

      a. Deadlock
      b. Starvation
      c. Livelock

---

3. What is starvation, give an example?

   Starvation is where the system allocates resources according to some policy such that progress is being made, however one or more processes never receive the resources they require as a result of that policy.

   Example, a printer that is allocated based on "smallest print job first" in order to improve the response for small jobs. A large job on a busy system may never print and thus *starve*

---

4. Two processes are attempting to read independent blocks from a disk, which involves issuing a *seek* command and a *read* command. Each process is interrupted by the other in between its *seek* and *read*. When a process discovers the other process has moved the disk head, it re-issues the original *seek* to re-position the head for itself, which is again interrupted prior to the *read*. This alternate seeking continues indefinitely, with neither process able to read their data from disk. Is this deadlock, starvation, or livelock? How would you change the system to prevent the problem?

   It is livelock. Allow each process to lock the disk and issue both commands together mutually exclusively and then release the lock.

---

5. Describe four ways to *prevent* deadlock by attacking the conditions required for deadlock.

   - Mutual exclusion condition
     - Make the resource sharable, i.e. allow concurrent access to read-only files. However, in general some resources are not shareable and require mutual exclusion.
   - Hold and wait condition
     - Dictate only a single resource can be held at any time. Not really practical.
     - Require that all required resource be obtained initially. If a resource is not available, all held resources must be releases before trying again - prone to livelock.

- No preemption condition
  - Preempt the resource (take it away from the holder). Not always possible.
- Circular wait condition
  - Order the resources numerically and request them in numerical order.

---

6. Answer the following questions about the tables.

   a. Compute what each process still might request and display in the columns labeled "still needs".
   b. Is the system in a safe or unsafe state? Why?

      Safe, feasible schedule p1,p4,p5,p2,p3

   c. Is the system deadlocked? Why or why not?

      No. There are not process remaining after the feasible schedule p1,p4,p5,p2,p3

   d. Which processes, if any, are or may become deadlocked?

      None

   e. Assume a request from p3 arrives for (0,1,0,0)
      1. Can the request be safely granted immediately?

         No

      2. In what state (deadlocked, safe, unsafe) would immediately granting the request leave the system?

         Unsafe

      3. Which processes, if any, are or may become deadlocked if the request is granted immediately?

         p2, p3

available

r1	r2	r3	r4
2	1	0	0

	current allocation				**maximum demand**				**still needs**			
process	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2				
p2	2	0	0	0	2	7	5	0				
p3	0	0	3	4	6	6	5	6				
p4	2	3	5	4	4	3	5	6				
p5	0	3	3	2	0	6	5	2				

	current allocation				**maximum demand**				**still needs**			
process	r1	r2	r3	r4	r1	r2	r3	r4	r1	r2	r3	r4
p1	0	0	1	2	0	0	1	2	0	0	0	0
p2	2	0	0	0	2	7	5	0	0	7	5	0
p3	0	0	3	4	6	6	5	6	6	6	2	2

p4	2	3	5	4	4	3	5	6	2	0	0	2
p5	0	3	3	2	0	6	5	2	0	3	2	0

R3000 and assembly

7. What is a *branch delay*?

> The pipeline structure of the MIPS CPU means that when a jump instruction reaches the "execute" phase and a new program counter is generated, the instruction after the jump will already have been decoded. Rather than discard this potentially useful work, the architecture rules state that the instruction after a branch is always executed before the instruction at the target of the branch.

---

8. The goal of this question is to have you reverse engineer some of the C compiler function calling convention (instead of reading it from a manual). The following code contains 6 functions that take 1 to 6 integer arguments. Each function sums its arguments and returns the sum as a the result.

```
#include <stdio.h>

/* function protoypes, would normally be in header files */
int arg1(int a);
int arg2(int a, int b);
int arg3(int a, int b, int c);
int arg4(int a, int b, int c, int d);
int arg5(int a, int b, int c, int d, int e );
int arg6(int a, int b, int c, int d, int e, int f);

/* implementations */
int arg1(int a)
{
  return a;
}

int arg2(int a, int b)
{
  return a + b;
}

int arg3(int a, int b, int c)
{
  return a + b + c;
}

int arg4(int a, int b, int c, int d)
{
  return a + b + c + d;
}

int arg5(int a, int b, int c, int d, int e )
{
  return a + b + c + d + e;
}

int arg6(int a, int b, int c, int d, int e, int f)
{
  return a + b + c + d + e + f;
}
```

```
/* do nothing main, so we can compile it */
int main()
{
}
```

The following code is the disassembled code that is generated by the C compiler (with certain optimisations turned of for the sake of clarity).

```
004000f0 <arg1>:
  4000f0:       03e00008        jr      ra
  4000f4:       00801021        move    v0,a0

004000f8 <arg2>:
  4000f8:       03e00008        jr      ra
  4000fc:       00851021        addu    v0,a0,a1

00400100 <arg3>:
  400100:       00851021        addu    v0,a0,a1
  400104:       03e00008        jr      ra
  400108:       00461021        addu    v0,v0,a2

0040010c <arg4>:
  40010c:       00852021        addu    a0,a0,a1
  400110:       00861021        addu    v0,a0,a2
  400114:       03e00008        jr      ra
  400118:       00471021        addu    v0,v0,a3

0040011c <arg5>:
  40011c:       00852021        addu    a0,a0,a1
  400120:       00863021        addu    a2,a0,a2
  400124:       00c73821        addu    a3,a2,a3
  400128:       8fa20010        lw      v0,16(sp)
  40012c:       03e00008        jr      ra
  400130:       00e21021        addu    v0,a3,v0

00400134 <arg6>:
  400134:       00852021        addu    a0,a0,a1
  400138:       00863021        addu    a2,a0,a2
  40013c:       00c73821        addu    a3,a2,a3
  400140:       8fa20010        lw      v0,16(sp)
  400144:       00000000        nop
  400148:       00e22021        addu    a0,a3,v0
  40014c:       8fa20014        lw      v0,20(sp)
  400150:       03e00008        jr      ra
  400154:       00821021        addu    v0,a0,v0

00400158 <main>:
  400158:       03e00008        jr      ra
  40015c:       00001021        move    v0,zero
```

a. `arg1` (and functions in general) returns its return value in what register?
b. Why is there no stack references in `arg2`?
c. What does `jr ra` do?
d. Which register contains the first argument to the function?
e. Why is the `move` instruction in `arg1` after the `jr` instruction.
f. Why does `arg5` and `arg6` reference the stack?

a. `v0`

b. There are no local variables inside the function, so the compiler does not need space on the stack to store them.
c. It jumps (changes the program counter) to the address in the `ra` register. The `ra` register is set by a `jal` instruction to the address of the instruction after `jal`. Thus function calls can be implemented with `jal` and `jr ra` instructions.
d. `a0`
e. The instruction after a jump (i.e. the instruction in the *branch delay slot* is executed prior to arriving at the destination of the jump. Thus, logically, the `move` instruction is executed before `arg1` returns.
f. Up to 4 arguments can be passed to a function in registers. Arguments beyond the fourth are passed on the stack?

---

9. The following code provides an example to illustrate stack management by the C compiler. Firstly, examine the C code in the provided example to understand how the recursive function works.

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

char teststr[] = "\nThe quick brown fox jumps of the lazy dog.\n";

void reverse_print(char *s)
{
  if (*s != '\0') {
    reverse_print(s+1);
    write(STDOUT_FILENO,s,1);
  }
}

 int main()
{
  reverse_print(teststr);
}
```

The following code is the disassembled code that is generated by the C compiler (with certain optimisations turned off for the sake of clarity).

a. Describe what each line in the code is doing.
b. What is the maximum depth the stack can grow to when this function is called?

```
004000f0 <reverse_print>:
  4000f0:       27bdffe8        addiu   sp,sp,-24
  4000f4:       afbf0014        sw      ra,20(sp)
  4000f8:       afb00010        sw      s0,16(sp)
  4000fc:       80820000        lb      v0,0(a0)
  400100:       00000000        nop
  400104:       10400007        beqz    v0,400124 <reverse_print+0x34>
  400108:       00808021        move    s0,a0
  40010c:       0c10003c        jal     4000f0 <reverse_print>
  400110:       24840001        addiu   a0,a0,1
  400114:       24040001        li      a0,1
  400118:       02002821        move    a1,s0
  40011c:       0c1000af        jal     4002bc <write>
  400120:       24060001        li      a2,1
  400124:       8fbf0014        lw      ra,20(sp)
  400128:       8fb00010        lw      s0,16(sp)
  40012c:       03e00008        jr      ra
  400130:       27bd0018        addiu   sp,sp,24
```

a.

```
004000f0 <reverse_print>:
  4000f0:       27bdffe8        addiu   sp,sp,-24
  Allocate 24 bytes on the stack, 16 for a0-a3 (unused) and 8 for ra
  and s0

  4000f4:       afbf0014        sw      ra,20(sp)
  Save the return address for the function on the stack. This
  function calls other functions, which means the ra register will be
  overwritten.

  4000f8:       afb00010        sw      s0,16(sp)
  Recall the 's' registers must be preserved when we return from this
  function. We only use s0, so save it on the stack so we can use the
  register in this function, but restore it before returning.

  4000fc:       80820000        lb      v0,0(a0)
  Load a character from the pointer passed as the first argument.

  400100:       00000000        nop
  nop

  400104:       10400007        beqz    v0,400124 <reverse_print+0x34>
  Test is the character is zero, if so, jump forward to 400124

  400108:       00808021        move    s0,a0
  This is on the delay slot, save the pointer in s0

  40010c:       0c10003c        jal     4000f0 <reverse_print>
  Call reverse print

  400110:       24840001        addiu   a0,a0,1
  This is in the delay slot, add 1 to the pointer to have reverse
  print start on the next character in the string.

  400114:       24040001        li      a0,1
  Load the file descriptor for write (1).

  400118:       02002821        move    a1,s0
  Remember s0 is preserved across function calls above, so s0 still
  contains the original pointer passed into the function. Pass the
  pointer to write.

  40011c:       0c1000af        jal     4002bc <write>
  Call write function

  400120:       24060001        li      a2,1
  Another delay slot, load the number of bytes write should output (1 byte).

  400124:       8fbf0014        lw      ra,20(sp)
  Restore the return address of this function in prep for return from
  function

  400128:       8fb00010        lw      s0,16(sp)
  Restore s0 to whatever it was before this function was called.

  40012c:       03e00008        jr      ra
  Return to the caller.

  400130:       27bd0018        addiu   sp,sp,24
  In the branch delay slot, deallocate the stack.
```

b. The stack of each invocation of `reverse_print` is 24 bytes, but the function is recursive. The allocation is 24 bytes times the length of the string, and thus if the string is unbounded, so is the recursion, and thus stack growth is also unbounded.

---

10. Why is recursion or large arrays of local variables avoided by kernel programmers?

    The kernel stack is usually a limited resource. A stack overflow crashes the entire machine.

Threads

11. Compare cooperative versus preemptive multithreading?

    Cooperative multithreading is where the running thread must explicitly yield() the CPU so that the dispatcher can select a ready thread to run next. Preemptive multithreading is where an external event (e.g. a regular timer interrupt) causes the dispatcher to be invoked and thus *preempt* the running thread, and select a ready thread to run next.

    Cooperative multithreading relies on the cooperation of the threads to ensure each thread receives regular CPU time. Preemptive multithreading enforces a regular (at least systematic) allocation of CPU time to each thread, even when a thread is uncooperative or malicious.

---

12. Describe *user-level threads* and *kernel-level threads*. What are the advantages or disadvantages of each approach?

    User-level threads are implemented in the application (usually in a "thread library"). The thread management structures (Thread Control Blocks) and scheduler are contained withing the application. The kernel has no knowledge of the user-level threads.

    Kernel threads are implemented in the kernel. The TCBs are managed by the kernel, the thread scheduler is the normal in-kernel scheduler.

    - User threads are generally faster to create, destroy, manage, block and activate (no kernel entry and exit required).
    - If a single user-level thread blocks in the kernel, the whole process is blocked. However, some libraries (e.g., most UNIX pthreads libraries) avoid blocking in the kernel by using non-blocking system call variants to emulate the blocking calls.
    - User threads don't take advantage of parallelism available on multi-CPU machines.
    - Kernel threads are usually preemptive, user-level threads are usually cooperative (Note: some user-level threads use alarms or timeouts to provide a tick for preemption).
    - User-level threads can be implemented on OSes without support for kernel threads.

---

13. A web server is constructed such that it is multithreaded. If the only way to read from a file is a normal blocking `read` system call, do you think user-level threads or kernel-level threads are being used for the web server? Why?

    A worker thread within the web server will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.

---

14. Assume a multi-process operating system with single-threaded applications. The OS manages the concurrent application requests by having a *thread* of control within the kernel for each process. Such a OS would have an in-kernel stack assocaited with each process.

    Switching between each process (in-kernel thread) is performed by the function `switch_thread(cur_tcb,dst_tcb)`. What does this function do?

    > The function saves the registers required to preserve the compiler calling convention (and registers to return to the caller) onto the current stack.
    >
    > The function saves the resulting stack pointer into the thread control block associated with `cur_tcb`, and sets the stack pointer to the stack pointer stored in destination tcb.
    >
    > The function then restores the registers that were stored previously on the destination (now current) stack, and returns to the destination thread to continue where is left off.
