Tutorial Week 2

Questions and Answers

Operating Systems Intro

1. What are some of the differences between a processor running in *privileged mode* (also called *kernel mode*) and *user mode*? Why are the two modes needed?

   In user-mode:
   + CPU control registers are inaccessible.
   + CPU management instructions are inaccessible.
   + Part of the address space (containing kernel code and data) are inaccessible.
   + Some device memory and registers (or ports) are inaccessible.

   The two modes of operation are required to ensure that applications (running in user-mode) cannot bypass, circumvent, or take control of the operating system.

---

2. What are the two main roles of an Operating System?

   1) It provides a high-level abstract machine for programmers (hides the details of the hardware)
   2) It is a resource manager that divides resources amongst competing programs or users according to some system policy.

---

3. Given a high-level understanding of file systems, explain how a file system fulfills the two roles of an operating system?

   At the level of the hardware, storage involves low-level controller hardware and storage devices that store blocks of data at many locations on the store. The OS filesystem abstracts above all these details and provides an interface to store, name and organise arbitrary unstructured data.

   The filesystem also arbitrates between competing processor by managing allocated and free space on the storage device, in addition to enforcing limits on storage consumption (e.g. quotas).

---

4. Which of the following instructions (or instruction sequences) should only be allowed in kernel mode?

   1. Disable all interrupts.
   2. Read the time of day clock.
   3. Set the time of day clock.
   4. Change the memory map.
   5. Write to the hard disk controller register.
   6. Trigger the write of all buffered blocks associated with a file back to disk (`fsync`).

   1,3,4,5 need to be restricted to kernel mode.

OS system call interface

5. The following code contains the use of typical UNIX process management system calls: `fork()`, `execl()`, `wait()` and `getpid()`. If you are unfamiliar with their function, browse the man pages on a UNIX/Linux machine get an overview. e.g man fork

   Answer the following questions about the code below.

   a. What is the value of `i` in the parent and child after `fork`
   b. What is the value of my_pid in a parent after a child updates it?
   c. What is the process id of `/bin/echo`?
   d. Why is the code after the `execl` not expected to be reached in the normal case?
   e. How many times is *Hello World* printed when `FORK_DEPTH` is 3?
   f. How many processes are created when running the code (including the first process)?

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

#define FORK_DEPTH 3

main()
{
  int i, r;
  pid_t my_pid;

  my_pid = getpid();

  for (i = 1; i <= FORK_DEPTH; i++) {

    r = fork();

    if (r > 0) {
      /* we're in the parent process after
         successfully forking a child */

      printf("Parent process %d forked child process %d\n",my_pid, r);

    } else if (r == 0) {

      /* We're in the child process, so update my_pid */
      my_pid = getpid();

      /* run /bin/echo if we are at maximum depth, otherwise continue loop */
      if (i == FORK_DEPTH) {
        r = execl("/bin/echo","/bin/echo","Hello World",NULL);

        /* we never expect to get here, just bail out */
        exit(1);
      }
    } else { /* r < 0 */
      /* Eek, not expecting to fail, just bail ungracefully */
      exit(1);
    }
  }
}
```

   a. The child is a new independent process that is a copy of the parent. `i` in the child will have whatever the value was in the parent at the point of forking.
   b. `my_pid` in the parent is not updated by any action of the child and the child and parent are independent after forking.
   c. `execl` replaces the *content* of a running process with specified executable. The process id does not change.
   d. A successful `execl` results in the current code being replaced. `execl` does not return if it succeeds as there is no previous code to return to.
   e. *Hello World* is printed 4 times if the `FORK_DEPTH` is 3.
   f. There are 8 processes involved in the execution of the code.

---

6. a. What does the following code do?
   b. In addition to `O_WRONLY`, what are the other 2 ways one can open a file?
   c. What open return in fd, what is it used for? Consider success and failure in your answer.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

main()
{
  int fd;
  int len;
  ssize_t r;

  fd = open("testfile", O_WRONLY | O_CREAT, 0600);
  if (fd < 0) {
    /* just ungracefully bail out */
    perror("Attempting to open fd");
    exit(1);
  }

  len = strlen(teststr);
  printf("Attempting to write %d bytes\n",len);

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  close(fd);

}
```

   a. The code writes a string to a file. It will create a new file if needed (`O_CREAT`).
   b. The other ways of opening a file are read-only (`O_RDONLY`) and read-write (`O_RDWR`).
   c. In case of failure `fd` is set to -1 to signify an error. In the case of success, `fd` is set to a *file descriptor* (an integer) that becomes a handle to the file. The file descriptor is used in the other file related systems cases to identify the file to operate on.

---

7. The following code is a variation of the previous code that writes twice.

   a. How big is the file (in bytes) after the two writes?
   b. What is `lseek()` doing that is affecting the final file size?
   c. What *errno* options are there in addition to `SEEK_SET`?

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

main()
{
  int fd;
  int len;
  ssize_t r;
  off_t off;

  fd = open("testfile2", O_WRONLY | O_CREAT, 0600);
  if (fd < 0) {
    /* just ungracefully bail out */
    perror("File open failed");
    exit(1);
  }

  len = strlen(teststr);
  printf("Attempting to write %d bytes\n",len);

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  off = lseek(fd, 5, SEEK_SET);
  if (off < 0) {
    perror("File lseek failed");
    exit(1);
  }

  r = write(fd, teststr, len);

  if (r < 0) {
    perror("File write failed");
    exit(1);
  }
  printf("Wrote %d bytes\n", (int) r);

  close(fd);

}
```

   a. 50 bytes. For each open file, the operating system keeps track of the current offset within the file. The current offset is where the next read or write will start from. The current offset is usually at the location of offset of the end of the previous read or write. So one would expect the file size to be 90 bytes after two 45 byte writes, except for `lseek`s interference (see below).
   b. `lseek` sets the current offset to a specific location in the file. The `lseek` in the code moves the current offset from 45 bytes (after the initial write) to 5 bytes from the start of the file. The second write begins from offset 5, writes 45 bytes, going 50 bytes in total in the file.
   c. See the man page for details on `EBDF_CUR` and `SEEK_END`

---

8. Compile either of the previous two code fragments on a UNIX/Linux machine and run `strace ./a.out` and observe the output.

   a. What is a `strace` doing?
   b. Without modifying the above code to print `fd`, what is the value of the file descriptor used to write to the open file?
   c. `printf` does not appear in the system call trace. What is appearing in it's place? What's happening here?

   a. `strace` is printing a trace of all system calls invoked by a process, together with the arguments to the system call. There are a lot of system calls at the beginning of a trace related to dynamically loading code libraries. Towards the end of the trace you will see the system calls you expect to see.
   b. 3
   c. `printf` is a library function that creates a buffer based on the string specification that it is passed. The buffer is then written to the console using `write()` to file descriptor 1.

---

9. Compile and run the following code.

   a. What do the following code do?
   b. After the program runs, the current working directory of the shell is the same. Why?
   c. In what directory does `/bin/ls` run in? Why?

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

main()
{
  int r;
  r = chdir("..");
  if (r < 0) {
    perror("Chdir");
    exit(1);
  }

  r = execl("/bin/ls","/bin/ls",NULL);
  perror("Double eek!");
}
```

   a. The code sets the current working directory of the process to be the parent directory (one higher in the directory hierarchy), and then runs `ls` to list the directory.
   b. The shell forks a child process that runs the code. Each process has its own current working directory, so the code above changes the current working directory of the child process, the current working directory of the parent process remains the same.
   c. `exec` replaces the content of the child process with `ls`, not the environment the child process runs in. The current working directory is part of the environment that the OS manages on behalf of every process, so `ls` runs in the current directory of child process.

---

10. On UNIX, which of the following are considered system calls? Why?

    1. `read()`
    2. `printf()`
    3. `memcpy()`
    4. `open()`
    5. `strcpy()`

    1 and 4 are system calls, 2 is a C library functions which can call `write()`, 3 and 5 a simply library functions.

Processes and Threads

11. In the *three-state process model*, what do each of the three states signify? What transitions are possible between each of the states, and what causes a process (or thread) to undertake such a transition?

    The three states are: *Running*, the process is currently being executed on the CPU; *Ready*, the process is ready to execute, but has not yet been selected for execution by the dispatcher; and *Blocked* where the process is not runnable as it is waiting for some event prior to continuing execution.

    Possible transitions are *Running to Ready*, *Ready to Running*, *Running to Blocked,* and *Blocked to Ready*.

    Events that cause transitions:

    + *Running to Ready:* timeslice expired, yield, or higher priority process becomes ready.
    + *Ready to Running:* Dispatcher chose the next thread to run.
    + *Running to Blocked:* A requested resource (file, disk block, printer, mutex) is unavailable, so the process is blocked waiting for the resource to become available.
    + *Blocked to Ready:* a resource has become available, so all processes blocked waiting for the resource now become ready to continue execution.

---

12. Given N threads in a uniprocessor system. How many threads can be *running* at the same point in time? How many threads can be *ready* at the same time? How many threads can be *blocked* at the same time?

    + Running threads = 0 or 1.
    + Blocked = N - Running - Ready
    + Ready = N - Running - Blocked

---

13. Compare reading a file using a single-threaded file server and a multithreaded file server. Within the file server, it takes 15 msec to get the request for work and do all the necessary processing, assuming the required block is in the main memory disk block cache. A disk operation is required for one third of the requests, which takes an additional 75 msec during which the thread sleeps. How many requests/sec can a server handled if it is single threaded? If it is multithreaded?

    In the single-threaded case, the cache hits take 15 msec and cache misses take 90 msec. The weighted average is 2/3 * 15 + 1/3 * 90. Thus the mean request takes 40 msec and the server can do 25 per second. For a multithreaded server, all the waiting for the disk is overlapped, so every request takes 15 msec, and the server can handle 66 2/3 requests per second.

Critical sections

14. The following fragment of code is a single line of code. How might a race condition occur if it is executed concurrently by multiple threads? Can you give an example of how an incorrect result can be computed for `x`.

```
x = x + 1;
```

    The single code statement is compiled into multiple machine instructions. The memory location corresponding to `x` is loaded into a register, incremented, and then stored back to memory. During the interval between the load and store in the first thread, another thread may perform a load, increment, and store, and then control passes back to the first thread, the results of the second are overwritten are them overwritten. Another outcome would be for the results of the first to be overwritten by the second (as the first thread loads, increments, then the second thread loads, increments, then the first thread stores, then the second thread stores).

---

15. The following function is called by multiple threads (potentially concurrently) in a multi-threaded program. Identify the critical section(s) that require(s) mutual exclusion. Describe the race condition or why no race condition exists.

```
int i;

void foo()
{
  int j;

  /* random stuff*/

  i = i + 1;
  j = j + i;

  /* more random stuff */
}
```

    There is no race condition on `j`, since it is a local variable per thread. However, `i` is a variable shared between threads. Thus `i = i + 1` would form a critical section (assuming no *random stuff* is dependent on `i`).

---

16. The following function is called by threads in a multi-thread program. Under what conditions would it form a critical section.

```
void inc_mem(int *iptr)
{
  *iptr = *iptr + 1;
}
```

    Whether `*iptr = *iptr + 1` forms a critical section depends on the scope of the pointer passed to `inc_mem`. If the pointer points to a local variable, then there is no race. If the pointer points to a shared global variable then there is potential for a race, and thus the increment would become a critical section.

Tutorial Week 3

estions and Answers

Synchronisation Problems

The following problems are designed to familiarise you with some of the problems that arise in concurrent programming and help you learn to identify and solve them.

Coordinating activities

What synchronisation mechanism or approach might one take to have one thread wait for another thread to update some state?

Semaphore style answer (semaphore `updated`, with count initialised a 0)

```
wait_for_update()
{
   P(updated)
}

signal_update_occurred()
{
   V(updated)
}
```

CV style answer with a variable `flag = 0`, a lock `l`, and a CV `cv` updated

```
wait_for_update()
{
  lock_acquire(l)
  while(flag == 0)
    cv_wait(cv,l)
  lock_release(l)
}

signal_update_occurred()
{
lock_acquire(l)
  flag = 1
  cv_signal(cv,l)
lock_release(l)
}
```

A particular abstraction only allows a maximum of 10 threads to enter the "room" at any point in time. Further threads attempting to enter the room have to wait at the door for another thread to exit the room. How could one implement a synchronisation approach to enforce the above restriction?

Semaphore style answer (semaphore `room`, with count initialised to 10)

```
enter_room()
{
   P(room);
}

leave_room()
{
   V(room);
}
```

CV style answer with a variable `occupants = 0`, a lock `room_lock`, and a CV `room_cv` updated

```
enter_room()
{
  lock_acquire(room_lock);
  while(occupants == 10)
    cv_wait(room_cv,room_lock);
  occupants = occupants + 1;
  lock_release(room_lock);
}

leave_room()
{
  lock_acquire(room_lock);
    occupants = occupants - 1;
    if (occupants == 9)
      cv_signal(room_cv,room_lock);
  lock_release(room_lock);
}
```

Multiple threads are waiting for the same thing to happen (e.g. a disk block to arrive from disk). Write pseudo-code for a synchronising and waking the multiple threads waiting for the same event.

A semaphore style answer with a variable `waiters_count = 0, disk_block_status = NOT_READY`, a semaphore `block_mutex` initialised to 1, and `block_sem` initialised to 0.

```
wait_block()
{
  P(block_mutex);
  if (disk_block_status == READY) {
    V(block_sem);
  } else {
    waiters_count = waiters_count + 1;
  }
  V(block_mutex);
  P(block_sem);
}

make_block_ready()
{
  P(block_mutex);
    disk_block_status = READY;
    while(waiters_count != 0) {
      waiters_count = waiters_count -1;
      V(block_sem);
    }
  V(block_mutex);
}
```

CV style answer with a variable `disk_block_status = NOT_READY`, a lock `block_lock`, and a CV `block_cv` updated

```
wait_block()
{
  lock_acquire(block_lock);
  while(disk_block_status != READY)
    cv_wait(block_cv,block_lock);
  lock_release(room_lock);
}

make_block_ready()
{
  lock_acquire(block_lock);
  disk_block_status = READY;
  cv_broadcast(block_cv,block_lock);
  lock_release(block_lock);
}
```

Identify Deadlocks

Here are code samples for two threads that use semaphores (count initialised to 1). Give a sequence of execution and context switches in which these two threads can deadlock.

Propose a change to one or both of them that makes deadlock impossible. What general principle do the original threads violate that causes them to deadlock?

```
semaphore *mutex, *data;

void me() {
        P(mutex);
        /* do something */

        P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}

void you() {
        P(data)
        P(mutex);

        /* do something */

        V(data);
        V(mutex);
}
```

The numbers inserted on the left indicate an execution order that results in deadlock.

```
semaphore *mutex, *data;

void me() {
1:      P(mutex);
        /* do something */

4:      P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}

void you() {
2:      P(data)
3:      P(mutex);

        /* do something */

        V(data);
        V(mutex);
}
```

To prevent deadlock, ensure the semaphores are acquired in the same order.

```
semaphore *mutex, *data;

void me() {
        P(mutex);
        /* do something */

        P(data);
        /* do something else */

        V(mutex);

        /* clean up */
        V(data);
}

void you() {
        P(mutex);
        P(data)

        /* do something */

        V(data);
        V(mutex);
}
```

More Deadlock Identification

Here are two more threads. Can they deadlock? If so, give a concurrent execution in which they do and propose a change to one or both that makes them deadlock free.

```
lock *file1, *file2, *mutex;

void laurel() {
        lock_acquire(mutex);
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(mutex);

        /* do something */

        lock_acquire(file1);

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}

void hardy() {
        /* do stuff */

        lock_acquire(file1);
        /* read from file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(file2);

        lock_acquire(mutex);
        /* do something */
        lock_acquire(file1);
        /* write to file 1 */
        lock_release(file1);
        lock_release(mutex);
}
```

The way to look for deadlock potential in this example is to record the order of locking for both threads and look for inconsistencies.

- laurel: mutex -> file1 -> file2; file2->file1
- hardy: file1 -> file2; mutex -> file1

Note file1 and file2 can be acquired in two different orders, which can results in deadlock. See the example below for how it can occur.

```
lock *file1, *file2, *mutex;

void laurel() {
1:      lock_acquire(mutex);
        /* do something */

2:      lock_acquire(file1);
        /* write to file 1 */

3:      lock_acquire(file2);
        /* write to file 2 */

4:      lock_release(file1);
7:      lock_release(mutex);

        /* do something */

8:      lock_acquire(file1);

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}

void hardy() {
        /* do stuff */

5:      lock_acquire(file1);
        /* read from file 1 */

6:      lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(file2);

        lock_acquire(mutex);
        /* do something */
        lock_acquire(file1);
        /* write to file 1 */
        lock_release(file1);
        lock_release(mutex);
}
```

The way to prevent deadlock is to define a global locking order and ensure all threads adhere to the order. For this example, lets choose mutex -> file1 -> file2. This implies we should adjust laurel. Here are two versions of laurel that result in deadlock freedom (there are others).

```
lock *file1, *file2, *mutex;

void laurel() {
        lock_acquire(mutex);
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(mutex);

        /* do something */

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}
```

Or

```
void laurel() {
        lock_acquire(mutex);
        /* do something */

        lock_acquire(file1);
        /* write to file 1 */

        lock_acquire(file2);
        /* write to file 2 */

        lock_release(file1);
        lock_release(mutex);

        /* do something */

        lock_release(file2);
        lock_acquire(file1);
        lock_acquire(file2);

        /* read from file 1 */
        /* write to file 2 */

        lock_release(file2);
        lock_release(file1);
}
```

Synchronised Lists

Describe (and give pseudocode for) a synchronised linked list structure based on thread list code in the OS/161 codebase (`kern/thread/threadlist.c`). You may use semaphores, locks, and condition variables as you see fit. You must describe (a proof is not necessary) why your algorithm will not deadlock.

In a general sense, the interface to the synchronised list is as follows.

```
init(list_t *);
add_head(list_t *list, node_t *node);
add_tail(list_t *list, node_t *node);
remove_head(list_t *list, node_t **node);
remove_tail(list_t *list, node_t **node);
insert_after(node_t *in_list, node_t *new_node);
insert_before(node_t *in_list, node_t *new_node);
remove(node_t *in_list);
```

Make sure you clearly state your assumptions about the constraints on access to such a structure and how you ensure that these constraints are respected.

In addition to a single lock solution, consider a solution involving a lock per node in the list. The instructive cases are `insert_after()` and insert_before(), and `remove()`

The thread subsystem in OS/161 uses a linked list of threads to manage some of its state (`kern/thread/threadlist.c`). This structure is not synchronised. Why not?

In OS/161, the threadlist is a doubling linked list where all the code is written under the assumption that is executes mutually exclusively, which in practice is ensured by interrupt disabling on the uniprocessor. Hence, there is no synchronisation in the threadlist code itself.

Writing a synchronised version of the general list management code could be done with a single lock that is acquired for all updates to the list and nodes itself. This would not allow independent updates to separate nodes in the list to proceed in parallel.

One could use a lock per node in the list in an attempt to enable more parallelism, combined with locking in only one list direction when multiple locks are require for a single update. This is not as simple as it seems, see the sample code for how complex individual locking becomes for some operations. Given the number of locks required, and that operations all start at the head to avoid deadline, fine-grained locking in this case is worse than a single lock as it simply add overhead of more locks to acquire.

```
#include
#include
#include
#include "list.h"

bool node_init(node_t *node)
{
        struct lock *lock;
        lock = lock_create("node lock");
        if (lock == NULL)
                return false;
        node->lock = lock;
        node->next = NULL;
        node->prev = NULL;
        return true;
}
void node_cleanup(node_t *node)
{
        lock_destroy(node->lock);
}

bool list_init(list_t *list)
{
        bool r;
        r = node_init(&list->head);
        if (! r)
                return false;
        r = node_init(&list->tail);
        if (! r) {
                node_cleanup(&list->head);
                return false;
        }
        list->head.next = &list->tail;
        list->tail.prev = & list->head;
        return true;
}


void list_cleanup(list_t *list)
{
        /* TBD */
        (void) list;
}

void add_head(list_t *list, node_t *node)
{
        insert_after(&list->head, node);
}

void add_tail(list_t *list, node_t *node)
{
        insert_before(list, &list->tail, node);
}

void insert_after(node_t *in_list, node_t *new_node)
{
        /* assumption, in_list is not removed before we successfully
           insert, this in only true for the head sentinal
           node */

        lock_acquire(new_node->lock); /* start new node in the locked sta
te */

        /* acquire in order head -> tail */
        lock_acquire(in_list->lock);
        lock_acquire(in_list->next->lock);

        /* now can add */
        new_node->next = in_list->next;
        new_node->prev = in_list;

        in_list->next = new_node;
        new_node->next->prev = new_node;

        lock_release(new_node->next->lock);
        lock_release(new_node->lock);
        lock_release(in_list->lock);
}



static bool hand_over_locking(list_t *list, node_t *target)
{
        /* do hand over locking until we find the target */
        /* Either we find the target and return with target and
           predecessor locked or we return false if we don't find
           it */

        node_t *p, *t;

        p = &list->head;
        lock_acquire(p->lock);

        t = p->next;
        lock_acquire(t->lock);
        while (t != target) {
                if (t == &list->tail) {
                        /* can't find target */
                        lock_release(p->lock);
                        lock_release(t->lock);
                        return false;
                }
                lock_release(p->lock);
                p = t;
                t = t-> next;
                lock_acquire(t->lock);
        }
        return true;
}


bool insert_before(list_t *list, node_t *in_list, node_t *new_node)
{
        /* assume in_list is not removed before starting, this is only
         true for the tail sentinel node */

        bool r;

        lock_acquire(new_node->lock); /* again start in locked state */


        /* lock acquire in order head -> tail

           This fragment attempt to avoid hand-over locking by starting a
t the
           tail. It accesses prev without holding the lock, so it might c
hange,
           and thus needs to check it's still prev after both locks are h
eld.


           p = in_list->prev;
           lock_acquire(p->lock);
           lock_acquire(in_list->lock);

           while (p->next != in_list) {

                  a node must have got inserted
                  prior to getting the lock

                 lock_release(p->lock);
                 lock_release(in_list->lock);
                 p = in_list->prev;
                 lock_acquire(p->lock);
                 lock_acquire(in_list->lock);
           }

           Yuck, this is livelock-able....  Basically one can't use the
           prev pointers to lock as we require the current node's lock
           prior to using prev, but then we can't lock the
           predecessor due to lock ordering needed to avoid deadlock.

        */


        /* we changed the prototype to include the list and use
           handover locking to lock in_list and predecessor */

        r = hand_over_locking(list, in_list);

        /* this should always succeed in a system that does not remove
           in_list */

        if (!r) {
                lock_release(new_node->lock);
                return r;
        }

        /* now have locks on prev and in_list */

        /* now can add */
        new_node->prev = in_list->prev;
        new_node->next = in_list;

        in_list->prev = new_node;
        new_node->prev->next = new_node;

        lock_release(new_node->prev->lock);
        lock_release(new_node->lock);
        lock_release(in_list->lock);
        return true;
}

static void remove_with_locks_held(node_t *in_list)
{
        in_list->prev->next = in_list->next;
        in_list->next->prev = in_list->prev;

}

void remove_head(list_t *list, node_t **node)
{
        node_t *n;
        lock_acquire(list->head.lock); /* sentinel head */
        lock_acquire(list->head.next->lock); /* removal candidate */

        if (list->head.next == &list->tail) {
                /* nothing to remove */
                lock_release(list->head.next->lock);
                lock_release(list->head.lock);
                *node = NULL; /* return null for empty list */
        }

        lock_acquire(list->head.next->next->lock); /* relies on lock
                                                      held above to use
                                                      the pointer */

        n = list->head.next;

        remove_with_locks_held(n);

        lock_release(list->head.lock); /* sentinel head */
        lock_release(n->lock); /* removed node */
        lock_release(list->head.next->lock); /* new head */

        *node = n;
}

void remove_tail(list_t *list, node_t **node)
{

/* leave this as an exercise, this is a handover involving three nodes
   where the last one is the tail */

        (void) list;
        (void) node;
}

bool remove(list_t *list, node_t *in_list)
{

        bool r;

        r = hand_over_locking(list, in_list);

        /* this should always succeed in a non-broken system,
           otherwise we have attempts to remove the same node more than
           once */

        if (!r)
                return r;

        /* in_list and predecessor is lock */

        lock_acquire(in_list->next->lock); /* grab the third lock */

        /* we can now remove the node */

        remove_with_locks_held(in_list);

        lock_release(in_list->prev->lock);
        lock_release(in_list->next->lock);
        lock_release(in_list->lock);
        return true;
}
```

Concurrency and Deadlock

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

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*

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.

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.

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

Tutorial Week 4

Questions and Answers

R3000 and assembly

1. 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.

---

2. 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?

---

3. 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.

---

4. 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

5. 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.

---

6. 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.

---

7. 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.

---

8. 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.

Kernel Entry and Exit

9. What is the `EPC` register? What is it used for?

   This is a 32-bit register containing the 32-bit address of the return point for the last exception. The instruction causing (or suffering) the exception is at `EPC`, unless `BD` is set in `Cause`, in which case `EPC` points to the previous (branch) instruction.

   It is used by the exception handler to restart execution at the at the point where execution was interrupted.

---

10. What happens to the `KUc` and `IEc` bits in the `STATUS` register when an exception occurs? Why? How are they restored?

    The 'c' (current) bits are shifted into the corresponding 'p' (previous) bits, after which `KUc = 0`, `IEc = 0` (kernel mode with interrupts disabled). They are shifted in order to preserve the current state at the point of the exception in order to restore that exact state when returning from the exception.

    They are restored via a `rfe` instruction (restore from exception).

---

11. What is the value of `ExcCode` in the `Cause` register immediately after a system call exception occurs?

    The value of `ExcCode` is 8.

---

12. Why must kernel programmers be especially careful when implementing system calls?

    System calls with poor argument checking or implementation can result in a malicious or buggy program crashing, or compromising the operating system.

---

13. The following questions are focused on the case study of the system call convention used by OS/161 on the MIPS R3000 from the lecture slides.
    1. How does the 'C' function calling convention relate to the system call interface between the application and the kernel?
    2. What does the most work to preserve the compiler calling convention, the system call wrapper, or the OS/161 kernel.
    3. At minimum, what additional information is required beyond that passed to the system-call wrapper function?

       a. The 'C' function calling convention must always appear to be adhered to after any system-call wrapper function completes. This invovles saving and restoring of the *preserved* registers (e.g., s0-s8, ra).

          The system call convention also uses the calling convention of the C-compiler to pass arguments to OS/161. Having the same covention as the compiler means the system call wrapper can avoid moving arguments around and the compiler has already placed them where the OS expects to find them.

       b. The OS/161 kernel code does the saving and restoring of preserved registers. The system call wrapper function does very little.

       c. The interface between the system-call wrapper function and the kernel can be defined to provide additional information beyond that passed to the wrapper function. At minimum, the wrapper function must add the system call number to the arguments passed to the wrapper function. It's usually added by setting an agreed-to register to the value of the system call number.

---

14. In the example given in lectures, the library function *read* invoked the `read` system call. Is it essential that both have the same name? If not, which name is important?

    System calls do not really have names, other than in a documentation sense. When the library function `read()` traps in to the kernel, it puts the number of the system call into a register or on the stack. This number is used to index into a jump table (e.g. a C *switch* statement). There is really no name used anywhere. On the other hand, the name of the library function is very important, since that is what appears in the program.

    Note: The kernel programmer may can the code in the operating system that implements `read` a similar name, e.g., `sys_read`. This is purely a convention for the sake of code clarity.

---

15. To a programmer, a system call looks like any other call to a library function. Is it important that a programmer know which library function result in system calls? Under what circumstances and why?

    As far as program logic is concerned it does not matter whether a call to a library function results in a system call. But if performance is an issue, if a task can be accomplished without a system call the program will run faster. Every system call involves overhead in switching from the user context to the kernel context. Furthermore, on a multiuser system the operating system may schedule another process to run when a system call completes, further slowing the progress in real time of a calling process.

---

16. Describe a plausible sequence of activities that occur when a timer interrupt results in a context switch.

    Note: In the table below, almost everything that is not the timer device or CPU is actually just code executing within the kernel. The distinction of "who" is there to clarify which kernel subsystem is notionally responsible.

Who	What
Timer Device	Raises interrupt line
CPU	Generates an interrupt exception
CPU	Switches from user to kernel mode
CPU	Begins executing the *Kernel Interrupt Handler*
Kernel Interrupt Handler	Changes the sp to the kernel stack for the interrupted process
Kernel Interrupt Handler	Saves the user registers for the interrupted process onto the stack
Kernel Interrupt Handler	Determines the source of the interrupt to be the timer device
Kernel Interrupt Handler	Calls the *Timer Interrupt Handler*
Timer Interrupt Handler	Acknowledges the interrupt to the timer device
Timer Interrupt Handler	Calls the *Scheduler* (dispatcher)
Scheduler	Chooses a new process to run
Scheduler	Calls the *Kernel* to switch to the new process
Kernel	Saves the current process's in-kernel context to the stack
Kernel	Switches to the new process's kernel stack by changing sp
Kernel	Reads the new process's in-kernel context off the stack
Kernel	Restores the user registers from the stack
Kernel	Sets the processor back to user mode
Kernel	Jumps to the new user process's PC

Tutorial Week 5

Questions and Answers

Memory Hierarchy and Caching

1. Describe the memory hierarchy. What types of memory appear in it? What are the characteristics of the memory as one moves through the hierarchy? How can do memory hierarchies provide both fast access times and large capacity?

   The memory hierarchy is a hierarchy of memory types composed such that if data is not accessible at the top of the hierarchy, lower levels of the hierarchy are accessed until the data is found, upon which a copy (usually) of the data is moved up the hierarchy for access.

   Registers, cache, main memory, magnetic disk, CDROM, tape are all types of memory that can be composed to form a memory hierarchy.

   In going from the top of the hierarchy to the bottom, the memory types feature decreasing cost per bit, increasing capacity, but also increasing access time.

   As we move down the hierarchy, data is accessed less frequently, i.e. frequently accessed data is at the top of the hierarchy. The phenomenon is called "locality" of access, most accesses are to a small subset of all data.

---

2. Given that disks can stream data quite fast (1 block in tens of microseconds), why are average access times for a block in milliseconds?

   Seek times are in milliseconds (e.g. .5 millisecond track to track, 8 millisecond inside to outside), and rotational latency (1/2 rotation) is in milliseconds (e.g. 2 milliseconds for 15,000rpm disk).

---

3. You have a choice of buying a 3 Ghz processor with 512K cache, and a 2 GHz processor (of the same type) with a 3 MB cache for the same price. Assuming memory is the same speed in both machines and is much less than 2GHz (say 400MHz). Which would you purchase and why? Hint: You should consider what applications you expect to run on the machine.

   If you are only running an small application (or a large one, that accesses only a small subset), then the 3GHz processor will be much faster. If you are running a large application access a larger amount of memory than 512K but generally less than 3MB, the 2GHz processor should be faster as the 3 GHz processor will be limited by memory speed.

Files and file systems

4. Consider a file currently consisting of 100 records of 400 bytes. The filesystem uses *fixed blocking*, i.e. one 400 byte record is stored per 512 byte block. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one record, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning, but there is room to grow at the end of the file. Assume that the record information to be added is stored in memory.

   a. The record is added at the beginning.
   b. The record is added in the middle.
   c. The record is added at the end.
   d. The record is removed from the beginning.
   e. The record is removed from the middle.
   f. The record is removed from the end.

	Contiguous	Linked	Indexed
a.	100r/101w	0r/1w	0r/1w
b.	50r/51w	50r/2w	0r/1w
c.	0r/1w	100r/2w	0r/1w
d.	0r/0w	1r/0w	0r/0w
e.	49r/49w	50r/1w 51r/1w	0r/0w
f.	0r/0w	99r/1w	0r/0w

---

5. In the previous example, only 400 bytes is stored in each 512 byte block. Is this wasted space due to internal or external fragmentation?

   Internal fragmentation

---

6. Old versions of UNIX allowed you to write to directories. Newer ones do not even allow the superuser to write to them? Why? Note that many unices allow you read directories.

   To prevent total corruption of the fs. eg `cat /dev/zero > /`

---

7. Given a file which varies in size from 4KiB to 4MiB, which of the three allocation schemes (*contiguous, linked-list, or i-node based*) would be suitable to store such a file? If the file is access randomly, how would that influence the suitability of the three schemes?

   Contiguous is not really suitable for a variable size file as it would require 4MiB to be pre-allocated, which would waste a lot of space if the file is generally mush smaller. Either linked-list of i-node-based allocation would be preferred. Adding random access to the situation (supported well by contiguous or i-node based), which further motivate i-node-based allocation to be the most appropriate.

---

8. Why is there VFS Layer in Unix?

   - It provides a framework to support multiple file system types concurrently without requiring each file system to be aware of other file system types.
   - Provides transparent access to all supported file systems including network file systems (e.g. NFS, CODA)
   - It provides a clean interface between the file system independent kernel code and the file system specific kernel code.
   - Provide support for *special* file system types like `/proc`.

---

9. How does choice of block size affect file system performance. You should consider both sequential and random access.

   - Sequential Access

     The larger the block size, the fewer I/O operations required and the more contiguous the disk accesses. Compare loading a single 16K block with loading 32 512-byte blocks.

   - Random Access

     The larger the block size, the more unrelated data loaded. Spatial locality of access can improve the situation.

---

10. Is the `open()` system call in UNIX essential? What would be the consequence of not having it?

    It is not absolutely essential. The `read` and `write` system calls would have to be modified such that:
    - The filename is passed in on each call to identify the file to operate on
    - With a file descriptor to identify the open session that is returned by `open`, the sycalls would also need to specify the offset into the file that the syscall would need to use.
    - Effectively opening and closing the file on each `read` or `write` would reduce performance.

---

11. Some operating system provide a *rename* system call to give a file a new name. What would be different compared to the approach of simply copying the file to a new name and then deleting the original file?

    The rename system call would just change the string of characters stored in the directory entry. A copy operation would result in a new directory entry, and (more importantly) much more I/O as each block of the original file is copied into a newly allocated block in the new file. Additionally, the original file blocks need de-allocating after the copy finishes, and the original name removed from the directory. A rename is much less work, and thus way more efficient than the copy approach.

---

12. In both UNIX and Windows, random file access is performed by having a special system call that moves the *current position* in the file so the subsequent `read` or `write` is performed from the new position. What would be the consequence of not having such a call. How could random access be supported by alternative means?

    Without being able to move the file pointer, random access is either extremely inefficient as one would have to read sequentially from the start each time until the appropriate offset is arrived at, or the an extra argument would need to be added to `read` or `write` to specify the offset for each operation.

Tutorial Week 7

Questions and Answers

Files and file systems

1. Why does Linux pre-allocate up to 8 blocks on a write to a file.

   Pre-allocating provides better locality when many writes to independent files are interleaved.

---

2. Linux uses a *buffer cache* to improve performance. What is the drawback of such a cache? In what scenario is it problematic? What alternative would be more appropriate where a buffer cache is inappropriate?

   The buffering writes in the buffer cache provides the opportunity for data to be lost if the system stops prior to the cache being flushed.

   Removable storage devices are particular problematic if users don't "unmount" them first.

   Robustness can be improved by using a write-through cache at the expense of poor write performance.

---

3. What is the structure of the contents of a directory? Does it contain attributes such as creation times of files? If not, where might this information be stored?

   - See lecture slides.
   - No, directories only have a name-to-inode mapping
   - Attributes of the file are stored in the inode itself.

---

4. The Unix inode structure contains a reference count. What is the reference count for? Why can't we just remove the inode without checking the reference count when a file is deleted?

   Inodes contain a reference count due to hard links. The reference count is equal to the number of directory entries that reference the inode. For hard-linked files, multiple directory entries reference a single inode. The inode must not be removed until no directory entries are left (ie, the reference count is 0) to ensure that the filesystem remains consistent.

---

5. Inode-based filesystems typically divide a file system partition into *block groups*. Each block group consists of a number of contiguous physical disk blocks. Inodes for a given block group are stored in the same physical location as the block groups. What are the advantages of this scheme? Are they any disadvantages?

   - Each group contains a redundant superblock. This make the file system more robust to disk block failures.
   - Block groups keep the inodes physically closer to the files they refer to than they would be (on average) on a system without block groups. Since accessing and updating files also involves accessing or updating its inode, having the inode and the file's block close together reduces disk seek time, and thus improves performance. The OS must take care that all blocks remain within the block group of their inode.

---

6. Assume an inode with 10 direct blocks, as well as single, double and triple indirect block pointers. Taking into account creation and accounting of the indirect blocks themselves, what is the largest possible number of block reads and writes in order to:

   a. Read 1 byte
   b. Write 1 byte

   Assume the inode is cached in memory.

   a. To write 1 byte, in the worst case:

      - 4 writes: create single indirect block, create double indirect block, create triple indirect block, write data block.
      - 3 reads, 2 writes: read single indirect, read double indirect, read triple indirect, write triple indirect, write data block
      - Other combinations are possible

   b. To read 1 byte, in the worst case:

      - 4 reads: read single indirect, read double indirect, read triple indirect, read data block

---

7. Assume you have an inode-based filesystem. The filesystem has 512 byte blocks. Each inode has 10 direct, 1 single indirect, 1 double indirect, and 1 triple indirect block pointer. Block pointers are 4 bytes each. Assume the inode and any block free list is always in memory. Blocks are not cached.

   a. What is the maximum file size that can be stored before
      1. the single indirect pointer is needed?
      2. the double indirect pointer is needed?
      3. the triple indirect pointer is needed?
   b. What is the maximum file size supported?
   c. What is the number of disk block reads required to read 1 byte from a file
      1. in the best case?
      2. in the worst case?
   d. What is the number of disk block reads and writes required to write 1 byte to a file
      1. in the best case?
      2. in the worst case?

   a.
      1. 5K
      2. 69K
      3. 8261K
   b. 1056837K
   c.
      1. 1
      2. 4
   d. What is the number of disk block reads and writes required to write 1 byte to a file
      1. 1w
      2. 4r/1w

---

8. A typical UNIX inode stores both the file's size and the number of blocks currently used to store the file. Why store both? Should not blocks = size / block size?

   Blocks used to store the file are only indirectly related to file size.

   - The blocks used to store a file includes and indirect blocks used by the filesystem to keep track of the file data blocks themselves.
   - File systems only store blocks that actually contain file data. Sparsely populated files can have large regions that are unused within a file.

---

9. How can deleting a file leave a inode-based file system (like ext2fs in Linux) inconsistent in the presence of a power failure.

   Deleting a file consists of three separate modifications to the disk:

   - Mark disk blocks as free.
   - Remove the directory entry.
   - Mark the i-node as free.

   If the system only completes a subset of the operations (due to power failures or the like), the file system is no longer consistent. See lecture slide for example of things that can go wrong.

---

10. How does adding journalling to a file system avoid corruption in the presence of unexpected power failures.

    Simply speaking, adding a journal addresses the issue by grouping file system updates into transactions that should either completely fail or succeed. These transactions are logged prior to manipulating the file system. In the presence of failure the transaction can be completed by replaying the updates remaining in the log.

Tutorial Week 8

Questions and Answers

Memory Management

1. Describe internal and external fragmentation.?

   External Fragmentation: total memory space exists to satisfy a request, but it is not contiguous.

   Internal Fragmentation: allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but is not being used.

---

2. What are the problems with multiprogrammed systems with fixed-partitioning?

   - Internal fragmentation.
   - Inability to run processes greater in size than a partition, but smaller then memory.

---

3. Assume a system protected with base-limit registers. What are the advantages and problems with such a protected system (compared to either a unprotected system or a paged VM system)?

   - Disadvantages
     - Partitions must be contiguous - external fragmentation.
     - Entire process must be in memory.
     - Cannot practically share memory with other processes.
   - Advantages
     - Applications are protected from each other
     - Memory for applications can be allocated dynamically and hardware translates the application (logical) addresses to allocated addresses.
     - Multiple concurrent executions of the same application is possible.
     - Compaction is also possible.

---

4. A program is to run on a multiprogrammed machine. Describe at which points in time during program development to execution time where addresses within the program can be bound to the actual physical memory it uses for execution? What are the implication of using each of the three binding times?

   - Compile/Link time binding.

     The executable itself contains the actual physical addresses it will use during execution. It can only run at one location, and only a single copy can run at a time, unless the executable is recompiled/relinked to a new location in physical memory prior to each execution.

   - Load time binding

     Addresses within the executable are annotated such that when the program is loaded into physical memory, the loader can bind the addresses to the correct location within physical memory as the program is loaded (copied) into memory. This process slows loading (increases startup latency), and increases executable file size (to hold annotations, minor point).

   - Run-time binding

     The compiler/linker generated addresses are bound to a logical address space (an abstract memory layout). The program executes using the logical addresses independent of where it is loaded into physical memory. At run-time the logical addresses are translated to the appropriate physical addresses by specialised hardware on each memory reference.

     Run-time binding is the most flexible (depending of the translation hardware available, e.g. page VM MMU), but it incurs the cost of translation on every memory reference, which can be significant.

---

5. Describe four algorithms for allocating regions of contiguous memory, and comment on their properties.

   1. First-Fit

      Scan memory region list from start for first fit.

      Tends to skip over potentially many regions at the start of list.

   2. Next-Fit

      Scan memory region list from point of last allocation to next fit.

      Breaks up large block at end of memory with any reduction in searching.

   3. Best-Fit

      Pick the closest free region in the entire list.

      Tends to leave small unusable regions, and slower due to requirement of search the entire list.

   4. Worst-Fit

      Find the worst fit in the entire list

      Slower as it searches the entire list, fragmentation still an issue.

---

6. What is compaction? Why would it be used?

   Moving all the allocated regions of memory next to each other (e.g. to the bottom of memory) to free up larger contiguous free regions.

Virtual Memory

7. What is swapping? What benefit might it provide? What is the main limitation of swapping?

   Swapping is where a process is brought into main memory in its entirety, run for a while, and then put completely back on disk.

   Swapping allows the OS to run more programs than what would fit in memory if all programs remained resident in memory.

   Swapping is slow as it has to copy the entire program's in-memory image out to disk and back.

---

8. What is Paging?

   In brief: Paging is where main memory is divided into equal-sized chunks (frames) and the programs address space (virtual address space) is also divided up into matching-sized chunks (pages). Memory is transfered to and from disk in units of pages.

---

9. Why do all virtual memory system page sizes have to be a power of 2? Draw a picture.

   The lower bits of a virtual address is not translated and passed through the MMU to form a physical address.

---

10. What is a TLB? What is its function?

    A *translation lookaside buffer* is an associative cache of page table entries used to speed up the translation of virtual addresses to physical addresses.

11. Describe a two-level page table and how it is used to translate a virtual address into a physical address.

    See lecture slides in the virtual memory lecture.

---

12. Given a two-level page table (in physical memory), what is the average number of physical memory accesses per virtual memory access in the case where the TLB has a 100% miss ratio, and the case of a 95% hit ratio

    3

    1 * .95 + .05 * (1 + 2) = 1.1

---

13. What are the two broad categories of events causing page faults? What other event might cause page faults?

    Illegal memory references and access to non-resident pages.

    Page faults may be used to set reference and dirty bits on architectures that do not support them in hardware.

---

14. Translate the following virtual addresses to Physical Addresses using the TLB. The system is a R3000. Indicate if the page is mapped, and if so if its read-only or read/write.

    The EntryHi register currently contains `0x00000200`.

    The virtual addresses are `0x00028123`, `0x0008a7eb`, `0x0005cfff`,`0x0001c642`, `0x0005b888`, `0x00034000`.

TLB	
EntryHi	**EntryLo**
`0x00028200`	`0x0063f400`
`0x00034200`	`0x001fc600`
`0x0005b200`	`0x002af200`
`0x0008a100`	`0x00145600`
`0x0005c100`	`0x006a8700`
`0x0001c200`	`0x00a97600`

    See page 6-2 of the MIPS R3000 Hardware Guide for revising TLB operation.

Results		
VAddr	**PhysAddr**	**Access**
`0x00028123`		Invalid
`0x0008a7eb`		Invalid (ASID mismatch)
`0x0005cfff`	`0x006a8fff`	Global bit, R/W
`0x0001c642`	`0x00a97642`	R/W
`0x0005b888`	`0x002af888`	Read-only
`0x00034000`	`0x001fc000`	R/W

---

15. Describe an inverted page table and how it is used to translate a virtual address into a physical address.

    See lecture slides in the virtual memory lecture.

---

16. Describe a hashed page table and how it is used to translate a virtual address into a physical address.

    See lecture slides in the virtual memory lecture.

---

17. Of the three page table types covered in lectures, which ones are most appropriate for large virtual address spaces that are sparsely populated (e.g. many single pages scattered through memory)?

    The 2-level suffers from internal fragmentation of page table nodes themselves. The IPT and HPT is best as it is searched via a hash, and not based on the structure of the virtual address space.

---

18. What is temporal and spatial locality?

    Temporal locality: states that recently accessed items are likely to be accessed in the near future.

    Spatial locality: says that items whose addresses are near one another tend to be referenced close together in time.

---

1. What is the difference between the different MIPS address space segments? What is the use of each segment?

This question aims to have you refresh your understanding of the MIPS virtual address space layout. See the last 5 slides of the "Virtual Memory" lecture, or the user-manual on the class web site.

2. What functions exist to help you manage the TLB? Describe their use. (Hint: look in kern/arch/mips/include/tlb.h)

```
/*
 * MIPS-specific TLB access functions.
 *
 *   tlb_random: write the TLB entry specified by ENTRYHI and ENTRYLO
 *        into a "random" TLB slot chosen by the processor.
 *
 *        IMPORTANT NOTE: never write more than one TLB entry with the
 *        same virtual page field.
 *
 *   tlb_write: same as tlb_random, but you choose the slot.
 *
 *   tlb_read: read a TLB entry out of the TLB into ENTRYHI and ENTRYLO.
 *        INDEX specifies which one to get.
 *
 *   tlb_probe: look for an entry matching the virtual page in ENTRYHI.
 *        Returns the index, or a negative number if no matching entry
 *        was found. ENTRYLO is not actually used, but must be set; 0
 *        should be passed.
 *
 *        IMPORTANT NOTE: An entry may be matching even if the valid bit
 *        is not set. To completely invalidate the TLB, load it with
 *        translations for addresses in one of the unmapped address
 *        ranges - these will never be matched.
 */

void tlb_random(uint32_t entryhi, uint32_t entrylo);
void tlb_write(uint32_t entryhi, uint32_t entrylo, uint32_t index);
void tlb_read(uint32_t *entryhi, uint32_t *entrylo, uint32_t index);
int tlb_probe(uint32_t entryhi, uint32_t entrylo);
```

3. What macros are used to convert from a physical address to a kernel virtual address?

This question aims to refresh your understanding of the relationship between KSEG0 and physical memory.

```
/*
 * The first 512 megs of physical space can be addressed in both kseg0 and
 * kseg1. We use kseg0 for the kernel. This macro returns the kernel virtual
 * address of a given physical address within that range. (We assume we're
 * not using systems with more physical space than that anyway.)
 *
 * N.B. If you, say, call a function that returns a paddr or 0 on error,
 * check the paddr for being 0 *before* you use this macro. While paddr 0
 * is not legal for memory allocation or memory management (it holds
 * exception handler code) when converted to a vaddr it's *not* NULL, *is*
 * a valid address, and will make a *huge* mess if you scribble on it.
 */
#define PADDR_TO_KVADDR(paddr) paddr_to_kvaddr(paddr)
static inline vaddr_t
paddr_to_kvaddr(paddr_t paddr){
    return ((paddr) + MIPS_KSEG0);
}

#define KVADDR_TO_PADDR(vaddr) kvaddr_to_paddr(vaddr)
static inline paddr_t
kvaddr_to_paddr(vaddr_t vaddr){
    return ((vaddr) - MIPS_KSEG0);
}
```

4. What address should the initial user stack pointer be?

```
/*
 * The top of user space. (Actually, the address immediately above the
 * last valid user address.)
 */
#define USERSPACETOP  MIPS_KSEG0

/*
 * The starting value for the stack pointer at user level.  Because
 * the stack is subtract-then-store, this can start as the next
 * address after the stack area.
 *
 * We put the stack at the very top of user virtual memory because it
 * grows downwards.
 */
#define USERSTACK     USERSPACETOP
```

5. What are the entryhi and entrylo co-processor registers? Describe their contents.

See the start of the "Virtual Memory II" lecture for an overview of how these registers are used to load the TLB, and the content of various bit-fields within the register. The follow macros are defined in OS161 for setting and masking values written to or read from the Entry* registers.

```
/*
 * TLB entry fields.
 *
 * Note that the MIPS has support for a 6-bit address space ID. In the
 * interests of simplicity, we don't use it. The fields related to it
 * (TLBLO_GLOBAL and TLBHI_PID) can be left always zero, as can the
 * bits that aren't assigned a meaning.
 *
 * The TLBLO_DIRTY bit is actually a write privilege bit - it is not
 * ever set by the processor. If you set it, writes are permitted. If
 * you don't set it, you'll get a "TLB Modify" exception when a write
 * is attempted.
 *
 * There is probably no reason in the course of CS161 to use TLBLO_NOCACHE.
 */

/* Fields in the high-order word */
#define TLBHI_VPAGE   0xfffff000
/*      TLBHI_PID     0x00000fc0 */

/* Fields in the low-order word */
#define TLBLO_PPAGE   0xfffff000
#define TLBLO_NOCACHE 0x00000800
#define TLBLO_DIRTY   0x00000400
#define TLBLO_VALID   0x00000200
/*      TLBLO_GLOBAL  0x00000100 */
```

6. What do the as_* functions do? Why do we need as_prepare_load() and as_complete_load()?

These functions are used by OS/161 internally to manage the address spaces of processes. These functions are dependent on the data structures (e.g. region lists and page tables) used to book-keep virtual memory, and thus you have to implement them. A summary of them is available in the header file.

`as_prepare_load()` is required as normally code segments are mapped read-only. `as_prepare_load()` enable writing to the code segment while the OS loads the code associated with the process. `as_complete_load()` then removes write permission to the code segment to revert it back to read-only.

```
/*
 * Functions in addrspace.c:
 *
 *    as_create - create a new empty address space. You need to make
 *                sure this gets called in all the right places. You
 *                may find you want to change the argument list. May
 *                return NULL on out-of-memory error.
 *
 *    as_copy   - create a new address space that is an exact copy of
 *                an old one. Probably calls as_create to get a new
 *                empty address space and fill it in, but that's up to
 *                you.
 *
 *    as_activate - make curproc's address space the one currently
 *                "seen" by the processor.
 *
 *    as_deactivate - unload curproc's address space so it isn't
 *                currently "seen" by the processor. This is used to
 *                avoid potentially "seeing" it while it's being
 *                destroyed.
 *
 *    as_destroy - dispose of an address space. You may need to change
 *                the way this works if implementing user-level threads.
 *
 *    as_define_region - set up a region of memory within the address
 *                space.
 *
 *    as_prepare_load - this is called before actually loading from an
 *                executable into the address space.
 *
 *    as_complete_load - this is called when loading from an executable
 *                is complete.
 *
 *    as_define_stack - set up the stack region in the address space.
 *                (Normally called *after* as_complete_load().) Hands
 *                back the initial stack pointer for the new process.
 *
 * Note that when using dumbvm, addrspace.c is not used and these
 * functions are found in dumbvm.c.
 */
```

7. What does vm_fault() do? When is it called? vm_fault() is called for any virtual memory related faults, i.e. TLB misses, writes to read-only pages, and accesses to invalid pages. It is responsible for resolving the fault by either returning an error, or loading an appropriate TLB entry for the application to continue.
8. Assuming a 2-level hierarchical page table (4k pages), show for the following virtual addresses:
    1. The page number and offset;
    2. the translated address (after any page allocation); and
    3. the contents of the page table after the TLB miss.

The page table is initially empty, with no L2 pages. You may assume that the allocator returns frames in order, so that the first frame allocated is frame 0, then frames 1, 2, 3, etc.

Answers:

Start out by determining which bits of the virtual address are used for each component of the virtual address. The level 1 index is the 10 most significant bits (i.e. `vaddr >> 22`), the level 2 index is the next 10 bits for the top, i.e. `(vaddr >> 12) & 0x3ff`, the offset bits are the last 12 bits of the address, i.e. `vaddr & 0xfff`.

The components of all the virtual addresses are as follows.

Virtual Address	Level 1 Index	Level 2 Index	Offset
0x100008	0x0	0x100	0x008
0x101008	0x0	0x101	0x008
0x1000f0	0x0	0x100	0x0f0
0x41000	0x0	0x41	0x000
0x41b00	0x0	0x41	0xb00
0x410000	0x1	0x10	0x000

If we examine the level 1 indexes, we see that there are only two indexes used, i.e. there are only two pointers to level-2 page tables. Let's assume we allocate them in advance.

```
level1[0x0] = PADDR_TO_KVADDR(1 << 12);
level1[0x1] = PADDR_TO_KVADDR(2 << 12);
```

Note the frame numbers can be converted into addresses of the start of the frame by shifting. The level-2 page tables are walked by the OS, i.e. lie in KSEG0, and so need to be converted into addresses in that range. If you get the types correct, you can then index off the pointers in the level-1 page tables.

```
(level1[0x0])[0x100] = (3 << 12) | TLB_VALID | TLB_DIRTY
(level1[0x0])[0x101] = (4 << 12) | TLB_VALID | TLB_DIRTY
(level1[0x0])[0x41] = (5 << 12) | TLB_VALID | TLB_DIRTY
(level1[0x1])[0x10] = (6 << 12) | TLB_VALID | TLB_DIRTY
```

Thus the translations for the above page table are:

Virtual Address	Physical Address
0x100008	0x3008
0x101008	0x4008
0x1000f0	0x30f0
0x41000	0x5000
0x41b00	0x5b00
0x410000	0x6000

9. What C expressions would you use to set or reset the valid bit in a page table entry?

```
pte = pte | TLBLO_VALID /* set */
pte = pte & ~TLBLO_VALID /* reset */
```

Answer:

kprintf() uses a lock to serialise access to the console. If the lock blocks, it context switches, which will call as_activate and flush the TLB, ejecting any newly inserted entry. Hence, the system ends up in an infinite loop replacing the TLB entry, and then ejecting it via kprintf()

10. What C expression would you use to test if the valid bit is set?

```
if (pte & TLBLO_VALID)
```

Answer:

see Q8

11. How would you extract the 12-bit offset (bits 0 - 11) from the virtual address?

```
offset = vaddr & ~ TLBHI_VPAGE
```

Answer:

see Q8

12. How would you convert the 10 most significant bits (22-31) of a virtual address into an index?

```
index = vaddr >> 22
```

Answer:

alloc_kpages() and free_kpages()

13. How would you convert the next 10 most significant bits (12-21) into an index?

```
index = (vaddr << 10) >> 22
```

Answer:

Using a bump pointer.

14. How would you round down a virtual address to the base of the page?

```
vaddr = vaddr &  TLBHI_VPAGE
```

Answer:

free_kpage() can't sensibly return free memory to a bump pointer allocator (except in the rare case that the last memory allocated is the first free'd), so it simply loses memory that is passed to it.

Tutorial Week 10

Questions and Answers

Virtual Memory

1. What effect does increasing the page size have?

   - Increases internal fragmentation, and the unit of allocation is now greater, hence greater potential for un-used memory when allocation is rounded up. Practically speaking, this is not usually an issue, and round up an application to the next page boundary usually results in a relative small amount wasted compared to the application size itself.
   - Decreases number of pages. For a given amount of memory, larger pages result in fewer page table entries, thus smaller (and potentially faster to lookup) page table.
   - Increases TLB coverage as the TLB has a fixed number of entries, thus larger entries cover more memory, reducing miss rate.
   - Increases page fault latency as a page fault must load the whole page into memory before allowing the process to continue. Large pages have to wait for a longer loading time per fault.
   - Increases swapping I/O throughput. Given a certain amount of memory, larger pages have higher I/O throughput as the content of a page is stored contiguously on disk, giving sequential access.
   - Increases the working set size. Work set is defined as the size of memory associates with all pages accessed within a window of time. With large the pages, the more potential for pages to include significant amounts of unused data, thus working set size generally increases with page size.

---

2. Why is demand paging generally more prevalent than pre-paging?

   Pre-paging requires predicting the future (which is hard) and the penalty for making a mistake may be expensive (may page out a needed page for an unneeded page).

---

3. Describe four replacement policies and compare them.

   - Optimal
     - Toss the page that won't be used for the longest time
     - Impossible to implement
     - Only good as a theoretic reference point: The closer a practical algorithm gets to optimal, the better
   - FIFO
     - First-in, first-out: Toss the oldest page
     - Easy to implement
     - Age of a page is isn't necessarily related to usage
   - LRU
     - Toss the least recently used page
     - Assumes that page that has not been referenced for a long time is unlikely to be referenced in the near future
     - Will work if locality holds
     - Implementation requires a time stamp to be kept for each page, updated on every reference
     - Impossible to implement efficiently
   - Clock
     - Employs a usage or reference bit in the frame table.
     - Set to one when page is used
     - While scanning for a victim, reset all the reference bits
     - Toss the first page with a zero reference bit.

---

4. What is thrashing? How can it be detected? What can be done to combat it?

   Thrashing is where the sum of the working set sizes of all processes exceeds available physical memory and the computer spends more and more time waiting for pages to transfer in and out.

   It can be detected by monitoring page fault frequency.

   Some processes can be suspended and swapped out to relieve pressure on memory.

Multi-processors

5. What are the advantages and disadvantages of using a global scheduling queue over per-CPU queues? Under which circumstances would you use the one or the other? What features of a system would influence this decision?

   Global queue is simple and provides automatic load balancing, but it can suffer from contention on the global scheduling queue in the presence of many scheduling events or many CPUs or both. Another disadvantage is that all CPUs are treated equally, so it does not take advantage of hot caches present on the CPU the process was last scheduled on.

   Per-CPU queues provide CPU affinity, avoid contention on the scheduling queue, however they require more complex schemes to implement load balancing.

---

6. When does spinning on a lock (busy waiting, as opposed to blocking on the lock, and being woken up when it's free) make sense in a multiprocessor environment?

   Spinning makes sense when the average spin time spent spinning is less than the time spent blocking the lock requester, switching to another process, switching back, and unblocking the lock requester.

---

7. Why is preemption an issue with spinlocks?

   Spinning wastes CPU time and indirectly consumes bus bandwidth. When acquiring a lock, an overall system design should minimise time spent spinning, which implies minimising the time a lock holder holds the lock. Preemption of the lock holder extends lock holding time across potentially many time slices, increasing the spin time of lock aquirers.

---

8. How does a read-before-test-and-set lock work and why does it improve scalability?

   See the lecture notes for code. It improves scalability as the spinning is on a read instruction (not a test-and-set instruction), which allows the spinning to occur on a local-to-the-CPU cached copy of the lock's memory location. Only when the lock changes state is cache coherency traffic required across the bus as a result of invalidating the local copy of the lock's memory location, and the test-and-set instruction which require exclusive access to the memory across all CPUs.

Scheduling

9. What do the terms *I/O bound* and *CPU bound* mean when used to describe a process (or thread)?

   The time to completion of a *CPU-bound* process is largely determined by the amount of CPU time it receives.

   The time to completion of a *I/O-bound* process is largely determined by the time taken to service its I/O requests. CPU time plays little part in the completion time of I/O-bound processes.

---

10. What is the difference between cooperative and pre-emptive multitasking?

    Cooperative the thread specifically release the CPU, pre-emptive the thread has no choice.

---

11. Consider the multilevel feedback queue scheduling algorithm used in traditional Unix systems. It is designed to favour IO bound over CPU bound processes. How is this achieved? How does it make sure that low priority, CPU bound background jobs do not suffer starvation?

    Note: Unix uses low values to denote high priority and vice versa, 'high' and 'low' in the above text does not refer to the Unix priority value.

    I/O bound processes typically have many short CPU bursts. If such a process is scheduled, it will typically not use up it's time slice. Priorities are recomputed taking into account the consumed CPU, and hence an I/O-bound process will end up having a higher priority than a process that started at the same priority level and which used more CPU cycles.

    Even a process with low priority will be scheduled eventually, since the priority of processes that are continually scheduled eventually also receive a low or lower priority.

    Also note that the recorded amount of CPU consumed (that is used to calculate priority) is aged (reduced) over time, and hence CPU-bound processes also increase in priority is they are not scheduled.

---

12. Why would a hypothetical OS always schedule a thread in the same address space over a thread in a different address space? Is this a good idea?

    Context switch is faster. Better locality. If done too often (always) it starves other tasks.

---

13. Why would a round robin scheduler NOT use a very short time slice to provide good responsive application behaviour?

    CPU is consumed when switching from one task to another. This switching does not contribute to application progress. If done very frequently (a very short time slice), a significant portion of available CPU will be wasted on scheduler overhead.

I/O

14. Describe *programmed I/O* and *interrupt-driven I/O* in the case of receiving input (e.g. from a serial port). Which technique normally leads to more efficient use of the CPU? Describe a scenario where the alternative technique is more efficient.

    For programmed I/O, the CPU waits for input by continuously reading a status register until it indicates input data is ready, afterwhich, the CPU can read the incoming data, and then return to reading the status register wainting for further input.

    For interrupt-driven I/O, the serial port device sends an interrupt to the CPU when data is ready to be read. The interrupt handler is then invoked, which acknowledges receiving the interrupt, reads the data from the device, and returns from the interrupt. The CPU is free for other activities while not interrupt processing.

    Programmed I/O is normally less efficient as the CPU is *busy waiting* for input when input is unavailable.

    Programmed I/O can be more efficient in circumstances where input it frequent enough such that the overhead of interrupts (getting into and out of the interrupt handler) is more than the average time spent busy waiting. A realistic example is fast networking where packets are nearly always available.

---

15. A device driver routine (e.g. `read_block()` from disk) is invoked by the file system code. The data for the filesystem is requested from the disk, but is not yet available. What do device drivers generally do in this scenario?

    They block on a synch primitive that is later woken up by the disk interrupt handler when the block is available.

---

16. Describe how I/O buffering can be formulated as a *bounded-buffer producer-consumer problem*.

    Take the a file system buffer cache as an example. File system writes are *produced* by application requests. The OS must select an available entry in the buffer cache (or re-use an existing one), or block until a free slot is available in the *bounded-size* buffer cache.

    The interrupt handler can be considered a consumer as after the write to disk completes, it marks the buffer cache entry as *clean* (consumed), freeing it up for further writes.

---

17. An example operating system runs its interrupt handlers on the kernel stack of the currently running application. What restriction does this place on the interrupt handler code? Why is the restriction required?

    The interrupt handler must not block waiting for a resource, as it indirectly blocks the application that was running.

    If the application has the resource the interrupt handler requires (e.g. memory buffers), the system is deadlocked.
