Solutions to Assignment 2

Programming Exercises

P1. The program hitme waits for a simple message to be sent by the program hit. When it receives the message, hitme writes "Ouch!" to standard output.

Currently, the hitme program receives its messages synchronously (i.e., it does nothing while it is waiting for messages to arrive). Your task is to modify the program so it receives these messages asynchronously. In your revised version, the program will run the juggle function while messages are not being received. You *may not* modify the juggle function.

Modify hitme.cc so that it uses pthread_create to create a thread that will listen for the messages.

Rather than deal with checking error returns from pthread_create, I added an exception-throwing wrapper function to the wrapper functions in wrappers.cc.

```
pthread_t epthread_create(const pthread_attr_t *attr,
                          void * (*start_func)(void *), void *arg) {
    pthread_t thread_id;
    int retval = pthread_create(&thread_id, attr, start_func, arg);

    if (retval == 0)
        return thread_id;
    else {
        errno = retval;
        throw system_error("pthread_create error");
    }
}
```

Next, I added a wrapper function (called start_message_receiver) that calls message_receiver. The program passes start_message_receiver to pthread_create rather than message_receiver, because pthread_create expects a function that takes a *void* * argument and returns a *void* * result, whereas message_receiver has no arguments and returns nothing. (Alternatively, I could have modified message_receiver, but this way I could leave message_receiver unchanged.)

The start_message_receiver function properly handles all exceptions. We have to be careful about exceptions whenever we pass a C++ function to a C function (such as an operating system library function). The C function does not understand exceptions, so the passed function (start_message_receiver in this case) must promise that it will not throw any exceptions. To fulfill this promise, the function must handle any exceptions that are thrown by the functions it calls. In C++ a function can show that it won't throw exceptions by having "**throw** ()" in its declaration.

```
void * start_message_receiver(void *arg) throw () {
    try {
        message_receiver();
        return 0;
    }
    catch (system_error &exn) {
        cerr << "system error: " << exn.what()
            << " (" << strerror(errno) << ")" << endl;
    }
    catch (exception &exn) {
        cerr << "exception raised: " << exn.what() << endl;
    }
    catch (...) {
        cerr << "mysterious exception raised" << endl;
    }
    return reinterpret_cast<void *>(−1);
}
```

Finally, I modified main to start the thread.

```
int main(int argc, char **argv)
{
    try {
        epthread_create(0, start_message_receiver, 0);
        juggle();
    }
    catch (system_error &exn) {
        cerr << "system error: " << exn.what()
            << " (" << strerror(errno) << ")" << endl;
    }
    catch (exception &exn) {
        cerr << "exception raised: " << exn.what() << endl;
    }
}
```

For bonus credit, create an alternate version that uses the Unix SIGIO or SIGPOLL facilities rather than threads (on systems supporting the BSD programming interfaces or the latest POSIX interfaces (NEXTSTEP and Linux), see the description of the FASYNC (or O_ASYNC) in your reference for the fcntl system call; on System V systems (Solaris), see the I_SETSIG ioctl in the streamio documentation). Because the LocalPort class hides implementation details, including the file descriptor it uses to receive messages, you will need to extend it. Compare the thread-based and SIGIO-based approaches and contrast their strengths and weaknesses.

To make using the SIGIO facilities easier, I extended the *LocalPort* class with two additional methods, setAsync() and poll(). The setAsync() method puts the port into asynchronous mode, causing SIGIO signals to be sent whenever a message arrives. The poll() method returns true if there is a message waiting, and false if there are no messages waiting.

```
void LocalPort::setAsync() {
        fcntl(socket_fd, F_SETOWN, getpid());
        fcntl(socket_fd, F_SETFL, FASYNC);
}

bool LocalPort::poll() {
        fd_set read_fds;
        struct timeval zero = { 0, 0 };

        FD_ZERO(&read_fds);
        FD_SET(socket_fd, &read_fds);

        return eselect(socket_fd+1, &read_fds, 0, 0, &zero);
}
```

In the hitme program, I replaced message_receiver with message_handler, which is called whenever a SIGIO signal is received. message_handler is defined as follows:

```
LocalPort myport(HIT_PORT_NAME);
RemotePortAddress sender;

void message_handler(int signo) throw () {
        try {
                char buffer[5];

                clog << "\t\t received signal that a message is waiting\n";
                while ( myport.poll() ) {
                        int n = myport.receive(sender, &buffer, sizeof(buffer));
                        if (n != 4)
                                throw logic_error("receive failed!? (size != 4)");
                        const char *const sender_name = sender.name();
                        clog << "\t\t message received (from " <<
                                (*sender_name ? sender_name : "<anonymous>")
                                << ", " << n << " bytes)" << endl;
                        cout << "Ouch!" << endl;
                }
                return;
        }
        catch (system_error &exn) {
                cerr << "system error: " << exn.what()
                        << " (" << strerror(errno) << ")" << endl;
        }
        catch (exception &exn) {
                cerr << "exception raised: " << exn.what() << endl;
        }
        catch (...) {
                cerr << "mysterious exception raised" << endl;
        }
        exit(2);
}
```

There are several things to note about this code. First, I again used **catch** to prevent exceptions escaping from inside the signal handler. This code is analogous to the **catch** blocks in start_message_handler, and exists for the same reason: Signal handlers aren't allowed to throw exceptions.

Second, notice the **while** loop involving poll(). The **while** loop is included to handle the case of several messages arriving in quick succession. When this happens, the program may receive fewer SIGIO signals than there are messages (because Unix only records that there is “at least one” outstanding SIGIO signal, not how many there are—Unix signals are like processor interrupts in this regard). To address this issue the SIGIO handler loops until poll returns false, ensuring that all waiting messages are dealt with (and that spurious SIGIO signals won’t cause problems).

Finally, I extended main to install the signal handler before it calls juggle. For convenience, I used sighandler, defined in the util module, to install the SIGIO handler.

```
int main(int argc, char **argv) {
        try {
                clog << "\t\t created server port (" << myport.name() << ")" << endl;
                sighandler(SIGIO, message_handler);
                clog << "\t\t installed SIGIO handler" << endl;
                myport.setAsync();
                clog << "\t\t set server port into asynchronous mode" << endl;
                juggle();
        }
        catch (system_error &exn) {
                cerr << "system error: " << exn.what()
                        << " (" << strerror(errno) << ")" << endl;
        }
        catch (exception &exn) {
                cerr << "exception raised: " << exn.what() << endl;
        }
}
```

An important difference between the SIGIO version and the Pthreads version is whether juggle runs while a message is being processed. In the SIGIO version, juggle is suspended while the message is handled, while in the Pthreads version it can continue to run.

The threading technique is cleaner and more general than the SIGIO technique:

- The SIGIO version requires that the *LocalPort* be declared as a global variable, whereas the Pthreads version used a local variable.
- If several parts of the program wanted to receive messages in the background, they could each create a port and a thread to listen on that port—this is more difficult to do using SIGIO because there can only be one SIGIO handler.
- If a program wanted to read a file in the background, it could use a thread in a similar way to the way it uses a thread to receive messages, but there is no analogous method involving SIGIO, since SIGIO is only available for “slow” devices like network connections and terminals.

Written Exercises

w1. What are *race conditions?* Explain, using the message-server and message-test programs from the project as examples.

Race conditions occur when multiple tasks manipulate a shared resource and the outcome is dependent on the relative timing of the tasks.

For example, consider two tasks using the original message server from the project: the first task fetches the value of variable x, increments the value, and stores it back to x on the server; the second task fetches the same variable from the server, decrements it, and stores it. Depending on how these tasks execute relative to one another, the final value of x could be unchanged, x+1, or x–1.

The sequence

```
a = fetch("x")
                              b = fetch("x")
++a
                              −−a
store("x", a)
                              store("x", a)
```

results in the increment being lost, whereas the sequence

```
                              b = fetch("x")
a = fetch("x")
                              −−a
++a
                              store("x", a)
store("x", a)
```

forgets about the decrement. The following execution sequence updates x correctly:

```
a = fetch("x")
++a
store("x", a)
                              b = fetch("x")
                              −−a
                              store("x", a)
```

W2. Explain why race conditions can still occur even if every data item is protected by a mutual-exclusion lock and the data item is always locked when it is read or written.

Mutual exclusion locks are meant to protect critical sections, yet the rule above does not say that the locks are held throughout the critical section, only that the lock is held when data is read or written. Thus, we can write the following code (analogous to the code for question W1) which is clearly suffering a race condition:

```
x_guard.lock()
a = x
x_guard.unlock()
                              x_guard.lock()
                              b = x
                              x_guard.unlock()
++a
                              ++b
x_guard.lock()
x = a
x_guard.unlock()
                              x_guard.lock()
                              x = b
                              x_guard.unlock()
```

In addition, even if critical sections are correctly protected, there can still be problems with deadlock. Suppose two tasks want to modify both x and y, both of which are protected by locks. The following code execution sequence will operate correctly:

```
x_guard.lock()
y_guard.lock()
old_x = x
old_y = y
x = old_y
y = old_x
y_guard.unlock()
x_guard.unlock()
                              y_guard.lock()
                              x_guard.lock()
                              new_y = x + y
                              new_x = x –y
                              x = new_y
                              y = new_x
                              x_guard.unlock()
                              y_guard.unlock()
```

but other execution orders may cause deadlock:

```
x_guard.lock()
                              y_guard.lock()
deadlock                      deadlock
```

Finally, if two critical sections are not commutative, there will be a race condition. For example, if x = 1, the execution order

```
x_guard.lock()
x = x + 2
x_guard.unlock()
                              x_guard.lock()
                              x = x * 2
                              x_guard.unlock()
```

results in x = 6, but the execution order

```
                              x_guard.lock()
                              x = x * 2
                              x_guard.unlock()
x_guard.lock()
x = x + 2
x_guard.unlock()
```

results in x = 4.

w3. Suppose we have a counter which we must increment atomically. Two solutions are proposed:

(a) Protect the variable using a simple spin lock. We assume an atomic test_and_set operation, where test_and_set(&*flag*) sets *flag* to be true and returns the value *flag* held just before it was set to true.

```
struct Counter {
        bool   mutex;
        int    value;

        void inc() {
                while (test_and_set(&mutex)) {
                        /* do nothing */
                }
                value = value + 1;
                mutex = false;
        }
}
```

(b) Use compare_and_swap, where compare_and_swap(&*var*, *oldval*, *newval*) atomically compares the contents of *var* with *oldval*, and if they are the same, updates *var* to hold *newval*. compare_and_swap returns the original value it found in *var*. (In other words, *oldval* is our "guess" for the value of the variable *var* (it is only a guess because some other process might modify *var* "behind our backs"). If our guess is right, compare_and_swap will modify *var* to hold *newval* and return *oldval*. If our guess is wrong, it will return the value that is really in *var*, which we can then use as a guess if we want to try again.)

```
struct Counter {
        int    value;

        void inc() {
                int old_value, swap_value, new_value;

                swap_value = value;
                do {
                        old_value = swap_value;
                        new_value = old_value + 1;
                        swap_value = compare_and_swap(&value, old_value, new_value);
                } while (swap_value != old_value);
        }
}
```

The second method saves the storage space of a boolean variable, since the class does not need a mutex field, but there is another reason why the second method is better. Determine and explain the problem that exists in (a) that does not exist in (b), and discuss whether replacing the spin lock with semaphores would address the problem.

The problem with (a) occurs when a task is preempted just after it obtains the mutex lock. No other process will be able to access the variable until the preempted task resumes and finishes its critical section. This could be a long time if the preempted task is a low-priority task.

The code for (b) does not use mutual exclusion, so it is unaffected by preemption.

Replacing the spin lock with semaphores may or may not solve the problem, depending on how semaphores are implemented. If the semaphore implementation addresses priority-inversion issues, then there will be no problem, otherwise there could still be a problem with a low-priority task holding the semaphore while a high-priority task waits.

Semaphores are likely to be less wasteful of processor resources, however, because even if a task does wait, it will be blocked rather than busy-waiting.

W4. Consider the following set of running processes, where the arrival time and the length of the next processing burst are given in milliseconds:

Process	Arrival Time	Burst Time	Priority (if applicable)
A	0	10	3
B	0	1	1
C	0	2	3
D	0	1	4
E	0	5	2

The processes are assumed to have arrived in the order A, B, C, D, E, all at time zero. This question concerns the behaviour of these processes under the following scheduling algorithms:

i. First Come, First Served (FCFS)

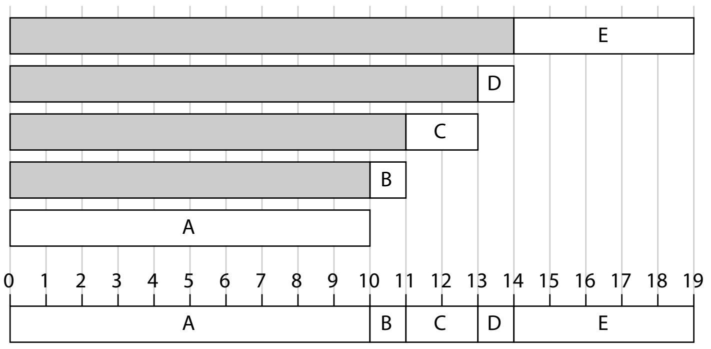
ii. Shortest Burst First (SBN (aka SJF))

iii. Nonpreemptive Priority (where zero is the highest priority)

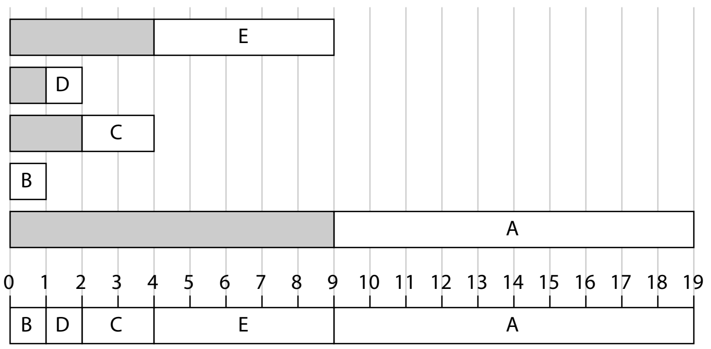
iv. Round Robin (RR) (where quantum = 1)

(a) Draw a Gantt chart illustrating the execution of these processes for each of the four scheduling algorithms given above.

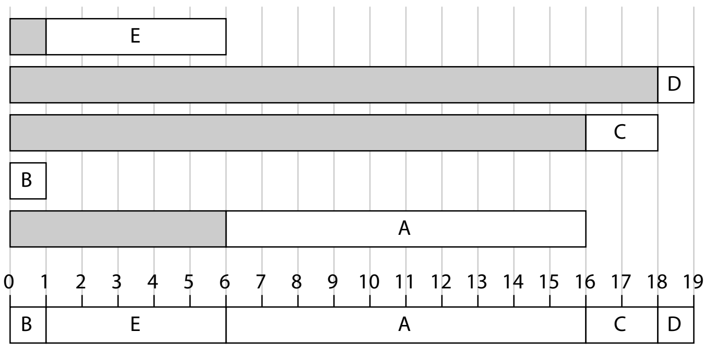
i.



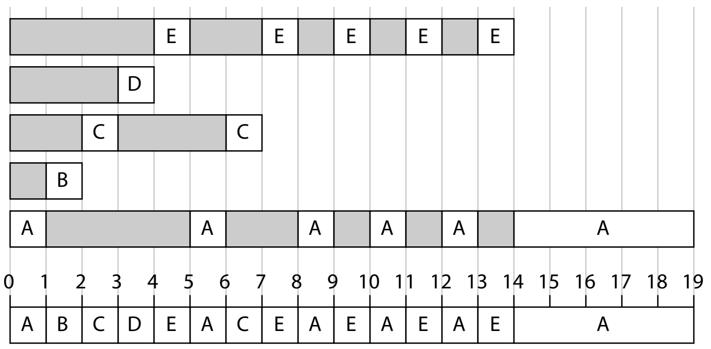
ii.



iii.



iv.



(b) What is the turnaround time of each process for each of the four scheduling algorithms?

	i	ii	iii	iv
A	10	19	16	19
B	11	1	1	2
C	13	4	18	7
D	14	2	19	4
E	19	9	6	14

(c) What is the waiting time of each process for each of the four scheduling algorithms?

	i	ii	iii	iv
A	0	9	6	9
B	10	0	0	1
C	11	2	16	5
D	13	1	18	3
E	14	4	1	9

(d) Which of the schedules results in the minimal average waiting time (over all processes)?

Shortest Burst First (with an average waiting time of 3.2).

w5. Although state diagrams for processes usually show a single blocked (or waiting) state, there is rarely a single queue of blocked processes associated with that state. Why?

There are multiple blocked queues because blocked processes are usually not all waiting for the same event. Usually there is a queue for every event (e.g., a queue for each semaphore, a queue for each incoming network connection, a queue for each terminal, etc.).

w6. Some multiprocessor systems have multiple ready queues, one for each processor, whereas other multiprocessor systems use a single shared ready queue. Give some of the tradeoffs involved in each design choice.

A single ready queue for all processors is a shared resource, with all the problems shared resources have. Thus the implementation must ensure that there are no race conditions if two processors try to access the queue at the same time. Even if the implementation is careful to avoid race conditions, there may be cache-performance issues with this design—if the ready queue is shared, each processor will not be able to store the queue in its cache, and will instead need to write it out to main memory.

Per-processor ready queues avoid contention problems, but they lead to the question of what should happen if one processor has an empty ready queue while another processor has a full ready queue. One option is to "steal" work from that other processor, but this approaches the synchronization problems we were trying to avoid. In practice, on a busy machine (where performance matters most) stealing is practical because steals are rare.

Per-processor ready queues also make it easier for tasks to be rescheduled on the same processor that they last used, and thereby benefit from any data that may be in the processor cache from their last burst.

Fairness, however, is easier to enforce with a single ready queue.

w7. Consider the following preemptive–priority-scheduling algorithm based on dynamically changing priorities. Larger priority numbers imply higher priority. When a process is waiting for the processor (in the ready queue, but not running), its priority changes at a rate α; when it is running, its priority changes at a rate β. All processes are given a priority of zero when they enter the ready queue. The parameters α and β can be set to give many different scheduling algorithms.

(a) What is the policy that results from $\beta > \alpha > 0$?

First-come first-served.

(b) What is the policy that results from $\alpha < \beta < 0$?

Preemptive last-come first-served.

(c) It might seem like updating the priorities of all the processes in the ready queue could take an amount of time that is dependent on the length of the ready queue (i.e., if the size of the ready queue is n, it would take $O(n)$ time), but in fact it is possible to do better. Explain how a clever implementation could make the time to update these priorities independent of the length of the ready queue.

In your scheme, would the following operations become dependent on the length of the ready queue, and if so, what would be the relationship?

i. Add a new process to the ready queue

ii. Switch from running one process to running another process

Express your answer using O notation if you know it. If these operations do not depend on the length of the ready queue, explain why.

We can restate the above scheduling algorithm in a way that makes it easy to implement constant time update of process priorities: There is a global clock, c, whose value *decreases* at a rate α. When the ready queue is empty, c is set to zero. When a process is waiting in the ready queue, its priority remains constant; when running, its priority changes at a rate β. All processes are given an initial priority equal to the current value of c when they enter the ready queue.

The priority numbers of processes are different, but the relative positions are the same. To calculate the priority number in the old scheme, simply subtract c.

This scheme requires that the ready queue be maintained as a priority queue, which typically requires $O(\log n)$ time to add a new process to the queue and to switch between processes if there are n processes in the ready queue.

(This solution assumes overflow is not an issue.)

(d) The algorithm above is underspecified—for example, it is unclear exactly when the priorities of processes are changed. Similarly, there are some cases that could occur in which it is not clear how the scheduler should behave. Explain one such case and specify an appropriate behaviour.

The algorithm does not state what happens when two processes have the same priority. For example, what happens when $\alpha = \beta$ and two processes arrive at the same time? Does the first to arrive run until it completes, or does the scheduler switch between them at every time quantum? Either would be acceptable policies.

(e) Can this algorithm provide a timesharing scheduling policy?

- If it *is* possible, state values of α and β that provide such a policy and explain the scheduling behaviour that will be observed—does it correspond to Round Robin, or is it different?

- If it *is not* possible, explain why and suggest an enhancement to the algorithm that would allow it to support timesharing scheduling policies.

Yes, it can. For example, $\alpha = 2, \beta = 1$ would allow a timesharing policy, albeit one where new tasks can be made to wait some time before they begin.

If we add the rule that the processor switches between processes of equal priority at every scheduling step, then $\alpha = \beta = 0$ would result in round robin—without this rule round robin is not possible.

w8. Contrast the following kernel designs (which may overlap):

(a) Monolithic nonreentrant kernel

(b) Reentrant kernel

(c) Multi-threaded kernel

(d) Microkernel

In a monolithic nonreentrant kernel, only one process may be running kernel code at a time. In this design, it is not easy for a thread to block while running kernel code, since the only way for it to block is for it to exit the kernel first. While this can be done (through a technique known as *lusering*—see *The Unix Hater's Handbook* for more details) it is complicated. In this design, interrupt service routines are very restricted as far as what kernel services they may use, but memory use is reduced because system only uses a single kernel stack.

Monolithic kernels lack modularity and clear interfaces between areas of functionality—the kernel is implemented as one large loosely structured program. The only way to add functionality to a monolithic kernel is to recompile it. The earliest kernels were monolithic nonreentrant kernels.

In a reentrant kernel, the idea of processes extends into the kernel and system calls are allowed to block inside the kernel. This reentrancy is usually achieved by having a kernel stack for each process, resulting in greater memory use than nonreentrant kernels. Allowing processes to block inside the kernel is *not* the same as allowing preemption inside the kernel.

A multithreaded kernel is a reentrant kernel which allows multiple tasks to be running inside the kernel. In this design, code running inside the kernel may be preempted and interrupt service routines are usually much freer in the services they can perform, often being run in a separate thread that is free to block. Solaris is an example of a multithreaded preemptable kernel, and is highly regarded for its efficiency and its scalability on multiprocessor machines.

A microkernel follows the principle of only putting the bare minimum of functionality into the kernel. The bulk of operating system services are implemented as user-level servers. For example, filesystem and networking facilities may be provided outside the kernel. Adding functionality to a microkernel is easy because it simply requires you to write a new server and tell people about it. And, because of the protection that exists between user-space tasks, a bug in the new server can't crash the existing servers. The idea behind microkernels is that they are simpler, easier to debug, and more suitable for distributed systems than traditional kernels. Unfortunately, the promise of ease of debugging has not been fully proven—the GNU Hurd operating system is based on a microkernel, yet it has been delayed due to difficulty with debugging the system. Microkernels also suffer slower performance than multithreaded kernels, because many more process switches are required to perform basic tasks (such as reading a file) than in regular kernels, due to the overheads of contacting several server processes.