Operating Systems – sprint 2022
Tutorial-Assignment 7

Instructor: Hans P. Reiser

Question 7.1: Interprocess Communication (IPC)

a. How can concurrent activities (threads, processes) interact in a (local) system?

Solution:
The two models of IPC are ***shared memory*** *and* ***message passing****. Concurrent activities can also communicate by other means. Some examples of communication facilities are:*

- *Shared memory*
  implicitly: *Same address space (e.g., another thread of the same process)*
  explicitly: *Shared memory area (e.g., between processes with different address spaces)*
- *OS communication facilities (messages (IPC), pipes, sockets)*
- *High-level abstractions (files, database entries)*

b. Asynchronous `send` operations require a buffer for sent but not yet received messages. Discuss possible locations for this message buffer and evaluate them.

Solution:
Message buffers can be located in the receiver's address space, in the kernel, or in the sender's address space.

receiver AS: *The receiver (e.g., some server process) can designate a finite region of its address space as a message queue. However, if the number of clients rises, the number of messages sent to this receiver might increase, thus requiring an ever larger receive area. If the receive area is not large enough, many IPCs will either fail and be repeated (doubling the IPC overhead) or block the sender, turning the asynchronous send undesirably into a synchronous one. Receive buffers do not scale well.*

kernel: *The kernel can allocate message buffers on demand for each asynchronous message that cannot be delivered instantly. However, malicious or bogus threads might flood the kernel with messages, thus we would need to impose limits on the message buffer size with consequences similar to the receiver-based buffers.*

Kernel-based buffers also scale poorly and can provide opportunities for denial-of-service attacks if not implemented properly.

sender AS: *Keeping the message in the sender's AS is a viable option. The sender has stored the to-be-sent message anyways, so we can use its current location as a message buffer. If we use the original message memory as message buffer, the application needs to be notified once the message has been consumed by the receiver so that the memory can be released or reused.*

Alternatively, we might provide a separate sent-message buffer and copy the message there iff an asynchronous send cannot deliver the message instantaneously. This approach would allow the application to overwrite the original message memory at the cost of requiring an (additional) copy of the message.

Sender-based buffering scales well, as only the sending processes pay for their communication requirements; applications that seldom communicate with other threads will never block due to insufficient buffer space.

Question 7.2: Race Conditions

a. Explain the term *race condition* with this scenario: Two people try to access a bank account simultaneously. One person tries to deposit 100 Euros, while another wants to withdraw 50 Euros. These actions trigger two update operations in a central bank system. Both operations run in "parallel" on the same computer, each represented by a single thread executing the following code:

```
current = get_balance();
current += delta;
set_balance(current);
```

where `delta` is either 100 or -50 in our example.

Solution:
A race condition is a situation where the correctness of a number of operations depends on the order in which the operations are executed. Let us assume that the initial balance is 1000 Euros. It might happen that the first thread copies the current balance (i.e., 1000*) into the variable* `current` *and is then preempted by the scheduler. The second thread starts, also copies the current balance (*1000*), withdraws 50 Euros, and updates the balance to 950 Euros. When the first thread resumes, it will add the deposit of 100 Euros to* `current` *and write the result back, so that the final balance will be 1100 Euros, but should actually be only 1050 Euros.*

The problem here is that reading the old balance, updating it, and writing it back is not atomic. If it was, either thread 1 or thread 2 would finish its update before the other is allowed to modify the balance.

b. Determine the lower and upper bounds of the final value of the shared variable `tally` as printed in the following program:

```
const int N = 50;
int tally;


void total ()
{
    for( int i = 0; i < N; ++i )
        tally += 1;
}
```

```
int main ()
{
    tally = 0;

    #pragma omp parallel for
    for( int i = 0; i < 2; ++i )
        total();

    printf( "%d\n", tally );
    return 0;
}
```

Note: The #pragma tells the compiler to create code to run parallel threads for each loop step. Assume that threads can execute at any relative speed and that a value can only be incremented after it has been loaded into a register by a separate machine instruction.

Solution:
On casual inspection, it appears that `tally` *will fall into the range of* $50 \leq$ `tally` ≤ 100 *since between 0 and 50 increments could go unrecorded due to lack of mutual exclusion. The basic argument contends that running these two threads concurrently should not derive results lower than the results produced by executing these threads sequentially.*

Consider the following interleaved sequence of the load, increment, and store operations:

(a) Thread A loads the value of `tally` *and increments it, but then loses the processor (it has already incremented its register to 1, but has not yet stored this value back into* `tally`*).*

(b) Thread B loads the value of `tally` *(still zero) and performs 49 complete increment operations, losing the processor just after it has stored the value 49 into the shared variable* `tally`*.*

(c) *Thread A regains control long enough to perform its first store operation (replacing the previous value of 49 in `tally` with 1) but after this it is immediately forced to relinquish the processor again.*

(d) *Thread B resumes long enough to load 1 (the current value of `tally`) into its register, but then it too is forced to give up the processor (note that this was B's final load operation).*

(e) *Thread A is rescheduled, but this time it is not interrupted and runs to completion, performing its remaining 49 load, increment, and store operations, which results in setting the value of `tally` to 50.*

(f) *Thread B is rescheduled with only one increment and store operation to perform before it terminates. It increments its register value to 2 and stores this value as the final value of the shared variable.*

Some thought will reveal that a value of lower than 2 cannot occur. Thus the proper and a bit astonishing range of values for the shared variable `tally` is $[2, 100]$.

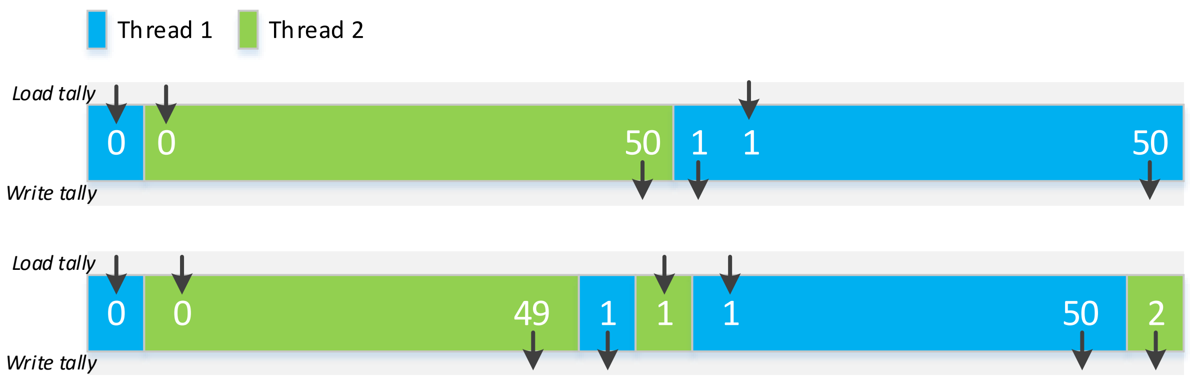

Abbildung 1: Execution timeline for the two threads in the worst case. The (wrong) intuitive solution at the top, the correct solution at the bottom.

c. Suppose that an arbitrary number $t > 2$ of parallel threads are performing the above procedure `total`. What (if any) influence does the value of t have on the range of the final values of `tally`?

Solution:
The range of final values of the shared variable `tally` is $[2, 50 \cdot t]$, *since it is possible for all other threads to be initially scheduled and run to completion in step (e) before thread B would finally destroy their work by finishing last.*

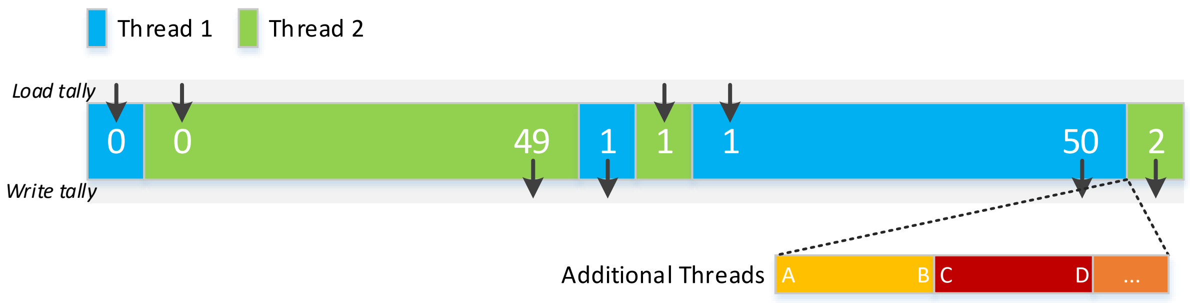

Abbildung 2: Execution timeline for $t > 2$ threads in the worst case.

d. Finally consider a modified `total` routine:

```
void total ()
```

```
{
    for( int i = 0; i < N; ++i )
    {
        tally += 1;
        sched_yield();
    }
}
```

What will be printed in the one-to-one model, when a voluntary yield is added?

Solution:
With the one-to-one model, `sched_yield` only causes a plethora of additional context switches, but does not guarantee that no thread is interrupted between *reading and writing `tally`. As a consequence, any value between 2 and 100 may be printed.*

Question 7.3: Critical Sections

a. Explain the term *critical section.*

Solution:
A critical section is a code region in which a thread accesses some common data such as a shared variable. As soon as one thread is executing inside a critical section, no other thread is allowed to execute the same critical section.

The remaining code after the exit section is called remainder section.

b. Enumerate and explain the requirements for a valid synchronization solution.

Solution:

Mutual Exclusion: *No two concurrent activities can enter a critical section (CS) that is protected by a synchronization primitive.*

Progress: *Threads running outside the critical section do not prevent threads in front of the CS (waiting at the synchronization primitive) from entering the CS.*

Bounded Waiting: *Bounded Waiting ensures that a thread waiting in front of a CS will eventually have the chance to enter the CS, that is, it ensures some "fairness" among threads that compete for the CS.*

c. Recap the banking example from the previous question. How could the race condition be avoided?

Solution:
Race conditions can be avoided if it is ensured that "critical" operations are performed atomically. To ensure atomicity, a synchronization primitive can be used, for example a lock, or a semaphore. A synchronization primitive ensures that at most one thread can be inside a critical section. Using a synchronization primitive `L`, we could fix the code from above:

```
lock(L);
current = get_balance();
current += delta;
set_balance(current);
unlock(L);
```

Now, even if one thread is preempted while updating the balance, the other cannot enter this critical code section, because it will have to wait (either actively or passively) at the *`lock(L)` instruction until the other thread performs the `unlock(L)` operation. Keep in mind that we have just passed on the problem to the lock function!*

Question 7.4: Pipes

a. Use cases for (anonymous) pipes: The `pipe` system call creates a pipe (in form of a pair of file descriptors) within the same process. For what kind of applications can you imagine that such an IPC channel can be useful? Do you know any application that you have recently used and that uses this IPC mechanism?

   Solution:
   The most important use case for such pipes are programms that launch other programs and want to communicate with those programs. One typical example that you surely have used is a shell. A shell may attach a pipe to the standard input and/or output of a program it launches for exchanging information with that program.

b. In assignment 6, in the Piazza forum, you have seen the command line command `(valgrind --tool=lackey --trace-mem=yes ls -la 2>&1 1>/dev/null) | python transform-lackey.py`
   Can you explain this command line, and how it relates to the IPC mechanism of pipes?

   Solution:
   The pipe symbol (|) in the command line directly corresponds to an anonymous pipe. The shell that parses this command will create a pipe (using the `pipe` system call) and then (using `fork`/`exec`) launch two subprocesses.

   - *The first processes will execute `valgrind`, with the write end of the pipe as its standard output.*
   - *The second process will execute `python` with the read end of the pipe attached to its standard input.*

   Standard input, output, and error are basically just files that have a specific file descriptor (0, 1, and 2). You can use the `dup2` system call to clone a file descriptor to a specific new value (e.g., you can use this to clone one end of the pipe to standard input or standard output). After a `fork()` system call, all open file descriptors are inherited by the child process.
   The command line makes use of this mechanism also within the first part: `2>&1` replaces standard error (2) with a clone of standard output (1), `1>/dev/null` opens the null device and replaces standard output with a clone of that file descriptor. As a result, the standard error output of valgrind will be redirected to the standard output, which is (due to the |) connected to the pipe.

c. Write a C program for Linux that creates two child processes, `ls` and `less`, and uses an ordinary pipe to redirect the standard output of `ls` to the standard input of `less`.

   Solution:

```
#define READ_END    0
#define WRITE_END   1

int main( void )
{
    int pid, pipefd[2];
    /* Create pipe. Parent has write and read end open*/
    int p = pipe(pipefd);
```

```
        if((pid = fork()) == 0) {
            /* first child, gets write end of pipe */
            dup2(pipefd[WRITE_END], STDOUT_FILENO); // Close stdout and replace
                                                    //  with write end.
            close(pipefd[WRITE_END]);               // One duplicated, one not needed
            close(pipefd[READ_END]);

            execlp("/bin/ls", "ls", NULL);          // Execute ls, which writes to
                                                    //  (replaced) stdout
        } else {
            /* parent, forks a second child */
            if((pid = fork()) == 0) {
                /* second child, gets read end of pipe */
                dup2(pipefd[READ_END], STDIN_FILENO); // Close stdin and replace
                                                      //  with read end
                close(pipefd[WRITE_END]);             // One duplicated, one not needed
                close(pipefd[READ_END]);

                execlp("/bin/less", "less", NULL);    // Execute less, which reads
                                                      //  from (replaced) stdin
            } else {
                /* parent, pipe fds must be closed so that 'less' gets an EOF */
                close(pipefd[READ_END]);
                close(pipefd[WRITE_END]);

                /* wait for both children to exit */
                wait(NULL);
                wait(NULL);
            }
        }
    }
```

Question 7.5: errno

a. When opening a file with the `open` system call, and similarly with many other system calls, the return value `-1` indicates an error. How can you find out why the system call failed? Write a C program for Linux that opens a file and in case of an error, prints a detailed error message explaining why opening the file failed.

Solution:

```
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>

// extern int errno ;

int main () {
    int errnum;
    int fd = open("testfile.txt", O_RDONLY);

    if (fd == -1) {
       fprintf(stderr, "Value␣of␣errno:␣%d\n", errno);
       perror("Error␣printed␣by␣perror");
       fprintf(stderr, "Error␣opening␣file:␣%s\n", strerror( errno ));
    } else {
        close (fd);
    }

    return 0;
}
```