COMP2046 Labs: Processes, Process Scheduling, Concurrency and Operating System APIs

Goals

The lab sessions focus on the use of **operating systems APIs** (specifically, the POSIX API in Linux) to create a set of processes. You will also use the APIs to influence the way in which the processes that you have created are scheduled by the **operating system's process scheduler**. What you aim to achieve during the lab sessions is similar to what was illustrated during the lectures, but this time for Linux.

You will use your code to investigate the influence of process scheduling in a multi-core/multi-CPU environment, in particular the influence of **hard and soft CPU affinity, process priorities, and the scheduling of your processes**as a function of the number of processes you have created. This will help you to better understand the principles explained in the lectures, and will illustrate the theory in practice. Whilst you will develop and run all code on the school servers, the observations are valid for most of the contemporary Linux systems (using the same process schedulers).

In summary, the key goals of the labs are:

- To illustrate the use of operating system APIs (necessary for the coursework)
- To help you understand process creation in Linux
- To help you better understand process scheduling
- To illustrate the practical relevance of the concepts discussed in the lectures
- To prepare you and give you the background knowledge that is required to successfully complete the coursework and the exam.

Background Information

- Additional information on Linux programming is available in the “Advanced Linux Programming” book (which is freely available online), amongst many others.
- Additional background information on process scheduling in Linux can be found online, including in the
  - tutorial listed here: link
  - the linux manual pages here: link

Writing code

Support for programming on the school servers, i.e. coding environments, is typically limited to emacs or vim, but you are more than welcome to use any of your own environments to develop the code, and compile it on the Linux servers using `gcc` program.

Creating/working with process

Background

There are several ways to create new processes on Linux, including fork() and clone(). fork() is the most commonly known (since it was also present in Unix) and the easiest approach to create new processes. However, it offers less flexibility than clone() (which enables to specify in detail which resources are cloned for the child process). When fork() is called by the parent process, it executes a system call to ask a service from the operating system, i.e., to create a new process, execute it, and carry out all internal administration that is required for that new process (e.g. creating the process control block and adding it to the process table). The fork() system call makes an exact copy of the current process. I.e., the child process will contain the exact same image as the parent process. After creation, the parent and child processes both continue with the first instruction immediately following the fork() call. Your code can distinguish between the parent and child process based on the value of the PID variable, which has not been set in the case of the child process. Hence, in the case of the child process, the variable will still contain the initial value. If, for some reason, the fork() call could not be carried out successfully, the PID returned to the parent process will be -1. The following example illustrates the use of fork(). More information on fork can be found here: link.

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

int main() {
    pid_t pid = 0;
    pid = fork();
    if(pid < 0) {
        printf("Could not create process\n");
        exit(1);
```

```
    } else if(pid == 0){
        sleep(1);
        printf("Hello from the child process\n");
    } else if(pid > 0) {
        printf("Hello from the parent process\n");
    }
    printf("This code will be executed by both the child and
parent\n");
}
```

Task 1

Using the same principles as above, write a program in which the parent process creates a pre-specified number of child processes (e.g. using a fork in a for loop). This number can be specified either on the command line or as a constant in your code (e.g. NUMBER_OF_PROCESSES). Make sure that you start with a relatively small number processes and assign every child process a unique index, e.g. between 1 and NUMBER_OF_PROCESSES (you will need this index later in these lab sessions). Add a printf statement to the child process that displays this ID together with the child's PID. Verify that your implementation is working as expected.

You may notice that more child processes are created than the number you specified (hence, why we asked you to start with a small number of processes to minimise the load on the school's servers). If this is the case, think of a way to resolve this (i.e. to create the exact number of processes) and implement it. You can increase the number of processes once you are confident that your code is working properly.

Make sure that you kill all child processes that you have created from the command line using, **killall -u XXXXX** , in which XXXXX is replaced by your own username (this will kill ALL your processes, including anything else that you have running). You can also use **killall programname** or **kill process_id** to kill particular process with its process ID.

Overwriting Process Images

Background

The `fork()` system call creates an exact copy of the parent process. The memory image of the child process can be overwritten using one of the `exec()` system calls, as illustrated below. More information on the different `exec()` system calls can be found here: link

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

int main() {
    int status;
    pid_t pid = fork();
    if(pid == -1) {
        printf("fork () error\n");
    } else if(pid == 0) {
        execl("/bin/ps", "ps", "l", 0);
        printf("This code should not run");
    }
}
```

The child process in the program above executes the `ps –l` command. The output will show quite a few different fields. Have a look at the meaning is of the individual entries in the man pages or on the web. Can you find the different process states and the process priorities discussed in the lectures in the output?

Task 2

Based on the example code above, modify your code from the previous task to create an additional child process for which you overwrite the “memory image” to run `ps –ejH`. (hint: you can use `execl("/bin/ps", "ps", "ejH", 0)` to overwrite the additional process's memory image);

Waiting for Processes

Background

After creation, child processes usually get a life of their own. On some occasions, you would want the parent process to wait for the child until it has finished (this is often also required for threads, as in the coursework). I.e. the parent process has to suspend its execution until the child process(es) have finished. This can be achieved by making the parent process execute a wait system call that takes the child's process identifier as one of the parameters. This is illustrated below. Related manual page can be found here: link

```
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_PROCESSES 4

int main() {
    int i, status;
    pid_t pid;
    printf("Hello from the parent process with PID %d\n",
getpid());
    pid = fork();
    if(pid < 0) {
        printf("fork error\n");
    } else if(pid == 0){
        sleep(1);
        printf("Hello from the child process with PID %d\n",
getpid());
        return;
    }
    waitpid(pid, &status, WUNTRACED);
    printf("Child process has finished\n");
}
```

Task 3

Modify your code above to ensure that the parent process (and only the parent process) waits for all child processes to finish before continuing (note that all child processes have to be able to run in parallel). When this is implemented, the command prompt should only display after all processes have finished, including the child processes. You should notice now that the prompt only appears after all the child processes have printed their messages.

General Thread Background

Threads, like processes, are a mechanism to allow a program to do more than one thing at a time. As with processes, threads appear to run concurrently; the Linux kernel schedules them asynchronously, interrupting each thread from time to time to give others a chance to execute.

Conceptually, a thread exists within a process.Threads are a finer-grained unit of execution than processes.When you invoke a program, Linux creates a new process and in that process creates a single thread, which runs the program sequentially.That thread can create additional threads; all these threads run the same program in the same process, but each thread may be executing a different part of the program at any given time.

When a program creates another thread, though, nothing is copied.The creating and the created thread share the same memory space, file descriptors, and other system resources as the original. If one thread changes the value of a variable, for instance, the other thread subsequently will see the modified value. Similarly, if one thread closes a file descriptor, other threads may not read from or write to that file descriptor.

Thread Creation

Each thread in a process is identified by a thread ID. When referring to thread IDs in C or C++ programs, use the type `pthread_t`.

Upon creation, each thread executes a thread function.This is just an ordinary function and contains the code that the thread should run. When the function returns, the thread exits. On Linux, thread functions take a single parameter, of type `void *`, and have a `void *` return type. The parameter is the thread argument: Linux passes the value along to the thread without looking at it. Your program can use this parameter to pass data to a new thread. Similarly, your program can use the return value to pass data from an exiting thread back to its creator. Related manual can be found here: link

The pthread_create function creates a new thread.You provide it with the following:

1. A pointer to a `pthread_t` variable, in which the thread ID of the new thread is stored.
2. A pointer to a thread attribute object. This object controls details of how the thread interacts with the rest of the program. If you pass `NULL` as the thread attribute, a thread will be created with the default thread attributes.
3. A pointer to the thread function. This is an ordinary function pointer, of this type: `void* (*) (void*)`
4. A thread argument value of type `void*`. Whatever you pass is simply passed as the argument to the thread function when the thread begins executing.

A call to pthread_create returns immediately, and the original thread continues executing the instructions following the call. Meanwhile, the new thread begins executing the thread function. Linux schedules both threads asynchronously, and your program must not rely on the relative order in which instructions are executed in the two threads.

```c
/*thread_creation.c*/

#include <pthread.h>
#include <stdio.h>

/* Prints x's to stderr. The parameter is unused. Does not return.
*/
void* print_xs (void* unused) {
    int i;
    for (i = 0; i < 2000; i++)
        fputc ('x', stderr);
    return NULL;
    }

/* The main program. */
int main () {
    pthread_t thread_id;
    /* Create a new thread. The new thread will run the print_xs
function. */
    pthread_create (&thread_id, NULL, &print_xs, NULL);
    /* Print o's continuously to stderr. */
    int i;
    for (i = 0; i < 2000; i++)
             fputc ('o', stderr);
    return 0;
}
```

Compile and link this program using the following code:

```
gcc -o thread_creation thread_creation.c -pthread
```

Try running it to see what happens. Notice the unpredictable pattern of x's and o's as Linux alternately schedules the two threads.

Passing Data to Threads

The thread argument provides a convenient method of passing data to threads. Because the type of the argument is `void*`, though, you can't pass a lot of data directly via the argument. Instead, use the thread argument to pass a pointer to some structure or array of data. One commonly used technique is to define a structure for each thread function, which contains the "parameters" that the thread function expects.

Using the thread argument, it's easy to reuse the same thread function for many threads. All these threads execute the same code, but on different data.

Joining Threads

Similar to `wait` in process creation, we have to force main to wait until the other two threads are done. What we need is a function similar to wait that waits for a thread to finish instead of a process. That function is `pthread_join`, which takes two arguments: the thread ID of the thread to wait for, and a pointer to a `void*` variable that will receive the finished thread's return value. If you don't care about the thread return value, pass `NULL` as the second argument.

The following program is similar to the previous example. This one creates two new threads, one to print x's and the other to print o's. Instead of printing infinitely, though, each thread prints a fixed number of characters and then exits by returning from the thread function. The same thread function, `char_print`, is used by both
threads, but each is configured differently using `struct char_print_parms`.
The main does not exit until both of the threads printing x's and o's have completed.

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

/* Parameters to print_function. */
```

```c
struct char_print_parms {
    /* The character to print. */
    char character;
    /* The number of times to print it. */
    int count;
};

 /* Prints a number of characters to stderr, as given by
PARAMETERS, which is a pointer to a struct char_print_parms. */

void* char_print (void* parameters) {
    /* Cast the cookie pointer to the right type. */
    struct char_print_parms* p = (struct char_print_parms*)
parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        fputc (p->character, stderr);
    return NULL;
}


/* The main program. */
int main()
{
    pthread_t thread1_id;
    pthread_t thread2_id;

    struct char_print_parms thread1_args;
    struct char_print_parms thread2_args;

    /* Create a new thread to print 3,000 'x's. */
    thread1_args.character = 'x';
    thread1_args.count = 3000;
    pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

    /* Create a new thread to print 2,000 o's. */
    thread2_args.character = 'o';
    thread2_args.count = 2000;
    pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

    /* Make sure the first thread has finished. */
    pthread_join (thread1_id, NULL);
```

```
    /* Make sure the second thread has finished. */
    pthread_join (thread2_id, NULL);

    return 0;
}
```

Thread Return Values (modified)

If the second argument you pass to pthread_join is non-null, the thread's return value will be placed in the location pointed to by that argument.The thread return value, like the thread argument, is of type `void*`. If you want to pass back a single int or other small number, you can do this easily by casting the value to `void*` and then casting back to the appropriate type after calling `pthread_join`.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/* Parameters to print_function. */
struct char_print_parms {
    /* The character to print. */
    char character;
    /* The number of times to print it. */
    int count;
};

 /* Prints a number of characters to stderr, as given by
PARAMETERS, which is a pointer to a struct char_print_parms. */

void* char_print (void* parameters) {
    /* Cast the cookie pointer to the right type. */
    struct char_print_parms* p = (struct char_print_parms*)
parameters;
    int i;
    for (i = 0; i < p->count; ++i)
        fputc (p->character, stderr);

    int *re_value = (int *)malloc(sizeof(int)); //modified
    *re_value = 100;
    return (void *) re_value;
}
```

```c
/* The main program. */
int main()
{
    pthread_t thread1_id;
    pthread_t thread2_id;

    int* getreturn;


    struct char_print_parms thread1_args;
    struct char_print_parms thread2_args;

    /* Create a new thread to print 3,000 'x's. */
    thread1_args.character = 'x';
    thread1_args.count = 3000;
    pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

    /* Create a new thread to print 2,000 o's. */
    thread2_args.character = 'o';
    thread2_args.count = 2000;
    pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

    /* Wait the first thread to complete, and get the returned
value. */
    pthread_join (thread1_id, (void*) &getreturn);
    /* Make sure the second thread has finished. */
    pthread_join (thread2_id, NULL);

    printf("\n Successfully get the returned value %d
\n",*getreturn);
    free(getreturn);
    return 0;
}
```

(Note : The fact that lines are occasionally printed through one another when the code is not synchronised. You could use semaphores to synchronise the writing of the processes/threads. I.e., allow only one process to write to the standard output at any point in time. This will avoid lines from showing up in random order.)

Task 4

The following code add sequential numbers from 1 to some maximum specified by the size of `ROWS` and `COLUMNS`. Modify the code and properly design the function, use multiple threads to work on the same problem and improve its calculation effenciency.

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

#define ROWS 4
#define COLUMNS 15000

int aiMatrix[ROWS][COLUMNS];
int iSum = 0;

void init() {
    int iRow, iColumn;
    int cont = 0;
    for(iRow = 0; iRow < ROWS; iRow++){
        for(iColumn = 0; iColumn < COLUMNS; iColumn++) {
            cont++;
            aiMatrix[iRow][iColumn] = cont;
        }
    }
}

int calculateSum(void * arg){
    int iRow = *((int *) arg);
    int iColumn;
    int Coltmp=0;
    for(iColumn = 0; iColumn < COLUMNS; iColumn++) {
           Coltmp = Coltmp + aiMatrix[iRow][iColumn];
    }
    return Coltmp;

}

int main(){
```

```
    init();
    int iRow;
    int Colsum[ROWS];

    for(iRow = 0;iRow < ROWS; iRow++){
        Colsum[iRow] = calculateSum(&iRow);
        iSum = iSum+Colsum[iRow];
    }

    printf("sum = %d\n", iSum);
    return 0;
}
```

* Different ways of passing and returning data between the main thread and multiple child threads (optional content)

In the modified version of the sample code of section **Thread returned value** tutorial, we have added a call to `malloc` to dynamically allocate memory for the result. This is important because memory allocated within a thread's stack will be deallocated when the thread exits, and accessing it outside the thread can lead to undefined behavior. Since all the threads share the same heap, we can use the shared memory (dynamically allocated memory) to pass the data between threads, note, don't forget to free the memory by using `free()` when finish using the dynamically allocated meomry.

There are three different ways of passing and return data between threads:

1. Passing and Returning a Single Value:
    - This method can be used when you need to pass a single value from the main thread to each child thread and get a result back.
    - Tutorial:
        - Define an array or a data structure to store the input values and results for each child thread.
        - Pass the address of the element corresponding to each child thread as the argument to the shared thread function.
        - Perform operations on the input value in the thread function and store the result in the respective element.
        - Access the results in the main thread after joining all child threads.
    - Sample Code:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
```

```c
#define NUM_THREADS 3

void* threadFunction(void* arg) {
    int inputValue = *((int*)arg); // Dereference the pointer

    // Perform operations on the input value or use it as needed

    int* result = (int*)malloc(sizeof(int));
    *result = inputValue * 2; // Example calculation

    pthread_exit((void*)result); // Exit the thread and return the
result
}

int main() {
    pthread_t threads[NUM_THREADS];
    int inputs[NUM_THREADS];
    void* threadResults[NUM_THREADS];

    // Set input values for each thread
    inputs[0] = 42;
    inputs[1] = 87;
    inputs[2] = 64;

    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, &threadFunction,
(void*)&inputs[i]) != 0) {
            fprintf(stderr, "Failed to create thread\n");
            return 1;
        }
    }

    // Wait for all threads to finish and get the results
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], &threadResults[i]) != 0) {
            fprintf(stderr, "Failed to join thread\n");
            return 1;
        }
    }

    // Access the results
    for (int i = 0; i < NUM_THREADS; i++) {
```

```
        int* result = (int*)threadResults[i];
        printf("Result #%d: %d\n", i+1, *result);
        free(result); // Free the dynamically allocated memory
    }

    return 0;
}
```

2. Passing and Returning a Structure:
    - This method is useful when you need to pass multiple values or a complex set of data between the main thread and each child thread, and get a result back.
    - Tutorial:
        - Define a structure that represents the data to be passed between threads, including fields for input and result.
        - Define an array or a data structure to store the input values and results for each child thread.
        - Pass the address of the element corresponding to each child thread as the argument to the shared thread function.
        - Perform operations on the input values in the thread function, store the result in the respective element.
        - Access the results in the main thread after joining all child threads.
    - Sample Code:

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

#define NUM_THREADS 3

struct ThreadData {
    int input;
    int result;
};

void* threadFunction(void* arg) {
    struct ThreadData* data = (struct ThreadData*)arg;
```

```c
    int inputValue = data->input;

    // Perform operations on the input value or use it as needed

    data->result = inputValue * 2; // Example calculation

    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    struct ThreadData threadData[NUM_THREADS];

    // Set input values for each thread
    threadData[0].input = 42;
    threadData[1].input = 87;
    threadData[2].input = 64;

    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, &threadFunction,
(void*)&threadData[i]) != 0) {
            fprintf(stderr, "Failed to create thread\n");
            return 1;
        }
    }

    // Wait for all threads to finish
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            fprintf(stderr, "Failed to join thread\n");
            return 1;
        }
    }

    // Access the results
    for (int i = 0; i < NUM_THREADS; i++) {
        printf("Result #%d: %d\n", i+1, threadData[i].result);
    }

    return 0;
}
```

3. Passing and Returning Dynamically Allocated Memory:
    - This method is useful when you need to pass large or dynamically created data structures between the main thread and each child thread, and get a result back.
    - Tutorial:
        - Allocate memory for each child thread using `malloc` to store the result.
        - Pass the pointer to the allocated memory as the argument to the shared thread function.
        - Perform operations on the input values in the thread function, store the result in the allocated memory.
        - Access the results in the main thread after joining all child threads.
        - Remember to free the allocated memory for each child thread when they are no longer needed to avoid memory leaks.
    - Sample Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 3

void* threadFunction(void* arg) {
    int* data = (int*)arg;

    // Perform operations on the data or use it as needed

    int* result = (int*)malloc(sizeof(int));
    *result = (*data) * 2; // Example calculation

    pthread_exit(result); // Exit the thread and return the result
}

int main() {
    pthread_t threads[NUM_THREADS];
    int* inputs = malloc(NUM_THREADS * sizeof(int));
    void* threadResults[NUM_THREADS];
```

```
    // Set input values for each thread
    inputs[0] = 42;
    inputs[1] = 87;
    inputs[2] = 64;

    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, &threadFunction,
(void*)&inputs[i]) != 0) {
            fprintf(stderr, "Failed to create thread\n");
            return 1;
        }
    }

    // Wait for all threads to finish and get the results
    for (int i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], &threadResults[i]) != 0) {
            fprintf(stderr, "Failed to join thread\n");
            return 1;
        }
    }

    // Access the results
    for (int i = 0; i < NUM_THREADS; i++) {
        int* result = (int*)threadResults[i];
        printf("Result #%d: %d\n", i+1, *result);
        free(result); // Free the dynamically allocated memory
    }

    free(inputs); // Free the dynamically allocated memory for
inputs

    return 0;
}
```

Named Semaphores (Process)

Background

A process semaphore, also known as a system semaphore or simply a semaphore, is a synchronization primitive used in concurrent programming. It allows multiple processes to coordinate their execution by controlling access to shared resources. Take for example, one could use semaphores to synchronise the writing of the processes. I.e., allow only one process to write to the standard output at any point in time. A full overview of using semaphores on Linux can be obtained by typing man sem_overview on the command line. Similar to shared memory, there are a number of steps that one has to go through to declare/access global semaphores (semaphores that are shared between processes – also called process semaphores):

- To work with process semaphores, we need to include the `<semaphore.h>` header.

```
#include <semaphore.h>
```

- Create a new named semaphore by calling `sem_open` and specifying the semaphore's name in the format `"/semaphore_name"` (make this unique). The second argument, `O_CREAT`, specifies that we want to create the semaphore if it doesn't already exist. The third argument, `0666`, sets the permissions for the semaphore. The fourth argument is the initial value of the semaphore. This function will return to a pointer that pointing to the semphore object.
- Use the semaphore by calling the functions `sem_post` and `sem_wait`:

```
sem_wait(mySemaphore); // Acquire the semaphore
// Critical section - protected by the semaphore,
The critical section represents the code that requires
exclusive access to shared resources. Only one process
can execute this code at a time.
```

```
sem_post(mySemaphore); // Release the semaphore
```

- Once we are done using the semaphore, it's important to close and unlink it using the `sem_close` and `sem_unlink` functions. Close the semaphore once the process no longer needs it calling `sem_close`. Deleting the semaphore from the system by calling `sem_unlink` (please do not forget to call this function since the semaphore may otherwise continue to exist in the OS, and the number of semaphores that are available is limited!)

```
sem_close(mySemaphore);   // Close the semaphore

sem_unlink("/semaphore_name");  // Unlink/remove the semaphore
```

A detailed description of how to use the different functions listed above can be found in the man pages, e.g. by typing "`man sem_open`"

```
/*semaphore_example.c*/
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <semaphore.h>
#include <stdio.h>

int main(int argc, char * argv[]){
    char * name = "my_semaphore";
    int VALUE = 2;
    sem_t * sema;
    //If semaphore with name does not exist, then create it with
VALUE
    printf("Open or Create a named semaphore, %s, its value is
%d\n", name,VALUE);

    sema = sem_open(name, O_CREAT, 0666, VALUE);
    //wait on semaphore sema and decrease it by 1
    sem_wait(sema);
    printf("Decrease semaphore by 1\n");

    /*...
    critical section
    ...*/
```

```
    //add semaphore sema by 1
    sem_post(sema);
    printf("Add semaphore by 1\n");
    //Before exit, you need to close semaphore and unlink it, when
all  processes have
    //finished using the semaphore, it can be removed from the
system       using sem_unlink
    sem_close(sema);
    sem_unlink(name);
    return 0;
}
```

Note that programs using the POSIX semaphores API must be compiled with -pthread to link against the real-time library. So you need compile semaphore.c like this:

```
gcc semaphore_example.c -pthread -o semaphore_example
```

Example

`NS-cars.c` and `EW-cars.c` are two programs that simulate traffic control. When North-South cars past the crossroads, they should ensure that road from north to south is available, if north-south road is not available, they keep waiting. This is similar to East-West cars.

Please check the following examples to see how the named semaphore is applied to provide mutual exclusion. You can also comment out semaphore related codes, run these two programs and see what happens.

```
/*NS-cars.c*/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
#include <unistd.h> // Add this header for sleep()

int main(int argc, char *argv[])
```

```
{
    int i,car=0, loop=5;
    sem_t *ns, *ew;
    ns = sem_open("ns", O_CREAT, 0666, 1);
    ew = sem_open("ew", O_CREAT, 0666, 0);
    for(i=0;i<loop;i++){
        sem_wait(ns);
        printf("Semaphore: The road from north to south is
open\n");
        sleep(1);
        printf("NS-Car: car %d passed\n",car++);
        sem_post(ew);
        sleep(2); // Add a longer delay between cars
    }
    printf("NS: Time is up. %d cars passed.\n",car);

    sem_close(ns);
    sem_close(ew);
    sem_unlink("ns");
    sem_unlink("ew");

    return 0;
}
```

```
/*EW-cars.c*/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

int main(int argc, char *argv[])
{
    int i,car=0, loop=3;
    sem_t *ns, *ew;

    ns = sem_open("ns", O_CREAT,0666,1);
    ew = sem_open("ew", O_CREAT, 0666, 0);
    for(i=0;i<loop;i++){
        sem_wait(ew);
        printf("Semaphore: The road from east to west is open\n");
        sleep(rand()%2+1);// Add a random delay between 1 to 2
```

```
seconds before each car passes
        printf("EW-Car: car %d passed\n",car++);
        sem_post(ns);
        sleep(2); // Add a longer delay between cars
    }
    printf("EW: Time is up. %d cars passed.\n",car);

    sem_close(ns);
    sem_close(ew);
    sem_unlink("ns");
    sem_unlink("ew");

    return 0;
}
```

Compile and run it with:

```
$ gcc NS-cars.c -pthread -o NS-cars
$ gcc EW-cars.c -pthread -o EW-cars
$ ./NS-cars & ./EW-cars &
```

Task 5

In this exercise, we will expand on the provided `NS-cars.c` and `EW-cars.c` programs to simulate traffic light control at an intersection. Your task is to modify the code to introduce a NS(NorthSouth-direction) traffic light and synchronize the movements of cars based on the state of the traffic light.

Unnamed Semaphore (Thread)

Background

Note that calling `sem_open` will result in a named semaphore, i.e., it has a name associated with it. This name can be shared between different processes, who can then "open" the semaphore (i.e. by retrieving a reference to the semaphore from the operating system) and use it to synchronise shared resources. The alternative to named semaphores are unnamed semaphores. These are declared in a region of memory that is shared between multiple threads (e.g. declared as a global variable or a variable in a region of shared memory – as above).

They differ in how they are created and destroyed, but otherwise work the same. Unnamed semaphores exist in memory only and require that processes have access to the memory to be able to use the semaphores. **This means they can be used only by threads in the same process or threads in different processes that have mapped the same memory extent into their address spaces.** Named semaphores, in contrast, are accessed by name and can be used by threads in any processes that know their names.

When we want to use POSIX semaphores within a single process, it is easier to use unnamed semaphores. This only changes the way we create and destroy the semaphore. To create an unnamed semaphore, we call the `sem_init()` function.

```
#include<semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

The `pshared` argument indicates if we plan to use the semaphore with multiple processes. If we just want to use the semaphore in a single process, then set it to zero. The `value` argument specifies the initial value of the semaphore.

Instead of returning a pointer to the semaphore like `sem_open()` does, we need to declare a variable of type `sem_t` and pass its address to `sem_init()` for

initialization. After initializing unnamed semaphore using `sem_init()` function, the `sem_wait()` and `sem_post()` functions can operate as usual.

When we are done using the unnamed semaphore, we can discard it by calling the `sem_destroy()` function.

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

After calling `sem_destroy()`, we can't use any of the semaphore functions with `sem` unless we reinitialize it by calling `sem_init()` again.

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;
sem_t unnamed_sema;

void* thread_function(void *arg)
{
    //wait on semaphore unnamed_sema and decrease it by 1 before
modify the shared resource
    sem_wait(&unnamed_sema);
    printf("Decrease semaphore by 1\n");

    counter += 1;
    printf("\n Job %d started\n", counter);
    sleep(2);
    printf("\n Job %d finished\n", counter);

    //add semaphore by 1 after modify the shared resource
    sem_post(&unnamed_sema);
    printf("Add semaphore by 1\n");
    return NULL;
}
```

```c
int main(void)
{
    int VALUE = 1;
    //initialize semaphore
    printf("Initialize unnamed semaphore, its value is
%d\n",VALUE);
    sem_init(&unnamed_sema, 0, VALUE);

    pthread_create(&(tid[0]), NULL, &thread_function, NULL);
    pthread_create(&(tid[1]), NULL, &thread_function, NULL);

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    //Before exit, you need to destroy semaphore
    sem_destroyed(&unnamed_sema);

    return 0;
}
```

Mutex (Thread)

Background

The thread sychronization can also be achieved by using Mutexes. A Mutex is a lock that we set before using a shared resource and release after using it. When the lock is set, no other thread can access the locked region of code. The Mutex lock will only be released by the thread who locked it. **The output should show all the child processes that you have created, their process state, and their priorities.** So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.

Similar to unnamed semaphore, a mutex is initialized and then a lock is achieved by calling the following two functions: `pthread_mutex_init` and `pthread_mutex_lock`. The mutex can be unlocked and destroyed by calling following functions : `pthread_mutex_unlock` and `pthread_mutex_destroy`.

Example

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* thread_function(void *arg)
{
    //mutex lock before modify the shared resource
    pthread_mutex_lock(&lock);
```

```
    counter += 1;
    printf("\n Job %d started\n", counter);
    sleep(2);
    printf("\n Job %d finished\n", counter);

    //mutex unlock after modify the shared resource
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void)
{
    //initialize mutex
    pthread_mutex_init(&lock, NULL);

    pthread_create(&(tid[0]), NULL, &thread_function, NULL);
    pthread_create(&(tid[1]), NULL, &thread_function, NULL);

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    //destroy mutex
    pthread_mutex_destroy(&lock);

    return 0;
}
```

Task 6

Modify the North-South crossroads example into a single process with multiple threads solution. Choose proper synchronisation method to ensure mutual exclusion.

Shared Memory

Background

Shared memory is a fast form of inter-process communication in which multiple processes write to the same (physical) memory, and hence, can share information with one another. Considering that multiple processes access the same memory, synchronisation must be considered and may have to be applied, e.g. using semaphores. Note that this is only required if multiple processes can access the exact same memory location simultaneously. Since shared memory is similar to accessing local memory, there is no performance penalty (unless synchronisation is required of course). Whilst the physical memory that is shared is the same (i.e. the frames), different process may have the physical memory segment attached to different logical addresses in their own address space (i.e. the pages).

Setting up shared memory consists of the following steps:

- Open a shared memory object using `shm_open`, specifying the name, optional flags, and the directory permissions, respectively
- Configure the size of the shared memory object using `ftruncate`, specifying the file descriptor, followed by size of the object.
- Map the shared memory object in to the processes' logical address space using `mmap`, specifying the address location at which to attach the memory, the size, the read/write protection, optional flags, the file descriptor for the shared object, and the offset (usually 0). Best practice is to specify NULL for the address location, thereby allowing Linux to decide itself where to attach the object into the logical address space, and returning the memory.
- Unlinking the shared memory segment using `shm_unlink`, specifying at least the name of the object

Additional information on any of the above functions can be easily found online, or in the man pages using, e.g. `man shm_open`. Note that you will have to specify `-lrt` on the command line to link in the appropriate library when compiling with

gcc.

In this example, we will design a program that uses shared memory to facilitate communication between two processes. The first process will write some data into the shared memory, and the second process will read that data.

1. Create a new C program file called `shared_memory_example.c`.
2. Include the required header files:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
```

3. Define a constant variable for shared memory name:

```
#define SHM_NAME "QZ123"
```

4. Implement the first process which writes data to shared memory:

```
void write_to_shared_memory() {
    // Open the shared memory object using shm_open()
    int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(1);
    }

    // Set the size of the shared memory object using ftruncate()
    if (ftruncate(shm_fd, sizeof(int)) == -1) {
        perror("ftruncate");
        exit(1);
    }

    // Map the shared memory object into the address space of the
current process using mmap()
    int* shared_data = (int*)mmap(NULL, sizeof(int), PROT_READ |
```

```
PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shared_data == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // Write some data to the shared memory location
    *shared_data = 12;

    // Unmap the shared memory object using munmap()
    if (munmap(shared_data, sizeof(int)) == -1) {
        perror("munmap");
        exit(1);
    }

    // Close the shared memory object using close()
    if (close(shm_fd) == -1) {
        perror("close");
        exit(1);
    }
}
```

5. Implement the second process which reads data from shared memory:

```
void read_from_shared_memory() {
    // Open the shared memory object using shm_open()
    int shm_fd = shm_open(SHM_NAME, O_RDONLY, 0666);
    if (shm_fd == -1) {
        perror("shm_open");
        exit(1);
    }

    // Map the shared memory object into the address space of the
current process using mmap()
    int* shared_data = (int*)mmap(NULL, sizeof(int), PROT_READ,
MAP_SHARED, shm_fd, 0);
    if (shared_data == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // Read the data from the shared memory location
```

```
    printf("Data read from shared memory: %d\n", *shared_data);

    // Unmap the shared memory object using munmap()
    if (munmap(shared_data, sizeof(int)) == -1) {
        perror("munmap");
        exit(1);
    }

    // Close the shared memory object using close()
    if (close(shm_fd) == -1) {
        perror("close");
        exit(1);
    }

    // Unlink the shared memory object using shm_unlink()
    if (shm_unlink(SHM_NAME) == -1) {
        perror("shm_unlink");
        exit(1);
    }
}
```

6. Create the main function and call the two processes:

```
int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork");
        exit(1);
    } else if (pid == 0) {
        // Child process
        write_to_shared_memory();
    } else {
        // Parent process
        read_from_shared_memory();
    }

    return 0;
}
```

7. Compile the program using the following command:

```
gcc -o shared_memory shared_memory.c -lrt
```

Task 7

Modify the sample code, allow the `main()` function uses `fork()` to create a child process. The parent process prompts the user for a message and writes it to shared memory using the `write_to_shared_memory()` function. The child process reads the message from shared memory using the `read_from_shared_memory()` function and displays it. (hint: you can use `fget()` to get user input, and `snprintf()` to write a string to the shared memory)

Note: The `wait()` system call is used in the parent process to wait for the child process to complete before printing "Parent process exiting.

General OS/Scheduler Background

Linux is a multi-tasking operating system. This means that, in practice, multiple processes run concurrently (by quickly alternating) or in parallel on multiple CPUs/cores. The operating system is responsible for managing these processes. This includes creating, destroying, context switching, and scheduling them, and the management of the execution traces (threads) and the resources they use. The processes you create during the lab sessions will contain only one thread. The reason for this is that the school's server offers more possibilities to influence the scheduling of processes than the scheduling of threads.

Every process in Linux is characterised by a unique and non-negative integer, called the process identifier (PID). The PID is used as an index into the process table where the associated process control block is stored. Note that the process control block in "Linux terminology" is called the task control block and that processes themselves are called tasks. Only a finite number of processes can exist simultaneously within the same system, and once the PIDs have reached their maximum value, they are wrapped around. Note that PID 1 is ``reserved'' for the init process (the parent of all processes in Linux).

As stated, the first process started on a Linux system is the init process. All other processes are created by the init process using system calls (similar to the ones you used above). The process scheduler is responsible for determining the order as well as the CPU/core on which the processes will run. In determining this order, the CPU scheduler takes a number of objectives and process characteristics into account (e.g. CPU bound process, I/O bound, priority, CPU affinity). The process scheduler also aims to get the best possible value for a number of objectives, including throughput, utilisation, fairness, and responsiveness. Note that Linux process schedulers have evolved considerably over different versions of Linux, with each approach having its own strengths and weaknesses.

Logging Process Times

Background

Like most programming languages, C can use the “system time”. This time can be used to check at what times/intervals the processes you created are using the CPU, e.g. relative to the start time of the parent process. The functions/data structures that you can use on Linux are illustrated below. Note that the returned time value contains two values: the number of seconds and the number of microseconds. Hence, in order to retrieve the number of milliseconds, you will have to manipulate the times.

```
#include <sys/time.h>
#include <stdio.h>
#include <unistd.h>

long int getDifferenceInMilliSeconds(struct timeval start, struct
timeval end);
long int getDifferenceInMicroSeconds(struct timeval start, struct
timeval end);

int main(){
    int i;
    struct timeval startTime, currentTime;
    gettimeofday(&startTime, NULL);
    sleep(1);
    gettimeofday(&currentTime,NULL);
    printf("Difference in milli-seconds
%d\n",getDifferenceInMilliSeconds(startTime,currentTime));
    printf("Difference in micro-seconds
%d\n",getDifferenceInMicroSeconds(startTime,currentTime));
}

long int getDifferenceInMilliSeconds(struct timeval start, struct
timeval end)
{
```

```
    int seconds = end.tv_sec - start.tv_sec;
    int useconds = end.tv_usec - start.tv_usec;
    int mtime = (seconds * 1000 + useconds / 1000.0);
    return mtime;
}

long int getDifferenceInMicroSeconds(struct timeval start, struct
timeval end)
{
    int seconds = end.tv_sec - start.tv_sec;
    int useconds = end.tv_usec - start.tv_usec;
    int mtime = (seconds * 1000000 + useconds);
    return mtime;
}
```

Task 8

- a. Extend the code developed in task 3 to estimate the time that it takes to fork a process, combined with the time it takes between forking the process and running the process (i.e., the response time). Note that these are only approximations since the parent process can be, for example, suspended between instructions. However, it should give you a reasonable estimate.
- b. Extend the code developed in task 4 to estimate the time that it takes to create a thread.

Task 9

Modify the code in task 3 to track the times at which the child processes are running, relative to the base time (the time at which the parent process starts). In other words, at the start of your parent process, you log what the current time is, and pass this timestamp on to all child processes who use it as "the base time" for logging their CPU activity. In each of the child processes, you retrieve the current time and take the difference between the current time and the start time of the parent process.

There are several steps in which you can achieve this (you are welcome to start with the last one if you feel confident you can do it):

- a. Define an infinite loop in the processes that prints the process ID and time at which the process was running (the one that you assigned yourself on the screen) in the following format:

  ```
  timevalue, process index
  timevalue, process index
  timevalue, process index
  ......
  ```

  You will probably see a fairly random pattern of IDs on the screen, however, when redirecting the output to a file and visualising it in a graph, patterns should start to emerge. A few warnings:

  - Make sure that you kill all child processes that you have created from the command line using, `killall –u XXXXX`, in which XXXXX is replaced by your own username (this will kill ALL your processes, including anything else that you have running).
  - None that, when re-directing output to a file, the file size will increase very rapidly! I.e., make sure that you don't leave your code running for too long, and that you delete the files afterwards! Or even better, run your code andredirect only the first 10000 lines to a file using `./task6a | head –n 10000 > output.csv`
  - Generate a visualisation that shows when processes are running (e.g. using Excel)

- b. Use a predefined duration for your experiments, i.e., make sure that the child processes terminate automatically when the maximum time (relative to the parent process) is exceeded.

- c. Replace the messages that you print on the screen with an array in which you log for each process the times that they were running. To avoid generating too much data (which becomes difficult to analyse), make sure that the child processes only run for a pre-specified and configurable

amount of time (see above, MAX_EXPERIMENT_DURATION set in milliseconds) and work with a granularity of milliseconds when logging at what times the child processes are running. Make sure you choose the data structure carefully so that:

  - You can log processes that are running in parallel
  - That you minimise any synchronisation requirements for this data-structure (in fact, choosing the correct data structure will prevent the need for synchronisation!)

  Make sure to write out the data once all child processes finish in the correct format. Try running your code with different numbers of child processes, e.g. 4 and 16, and assess what the impact is on the process behaviour and the scheduling of the processes.

  Note that your output may sometimes not be entirely what you expected. Occasionally, you will find that the output for multiple processes is printed out randomly and on other occasions you may notice that formatting is not entirely what you would hope for. This is due to multiple processes writing to the standard output at the same time. There are two ways out of this:

  - One would have to synchronise the writing (e.g. using semaphores) by allowing only one process to write to the standard output at any one time.
  - Alternatively, you could make sure that only the parent process writes the information out once all child processes have finished, however, this would require the use of shared memory between the parent and child process (i.e., the child process writes the values to the parent's memory).

- d. Modify your code to use shared memory. Make sure that you choose your data structure such that you do not have to enforce mutual exclusion/synchronisation. Make the parent process print out all the values that were written to the shared memory by the child processes.

- f. Modify the code in such a way that the individual child processes print out their timings. Use named semaphores to make sure that only one child

process prints to the standard output at any given point in time.

CPU Affinity

Background

In multi-core/multi-processor systems, processes/thread can run on different CPUs. Linux knows hard and soft CPU affinity. Under normal circumstances, soft affinity is used. I.e., processes can run on any available CPU/core, and migrate between CPUs to balance load. Hard affinity can be set explicitly, again by using system calls to ask the operating system's scheduler to run the process on the specified (set) of CPU(s) only. The use of hard affinity is illustrated below.

```
#define _GNU_SOURCE
#include <sched.h>

int main(){
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(4, &cpuset); // run on the 5th core (starts at 0)
    CPU_SET(5, &cpuset); // add core 6
    sched_setaffinity(getpid(), sizeof(cpu_set_t), &cpuset);
}
```

Task 10

Modify your code from task 9 to run on one single CPU/core/logical core (choose one of the 8 cores at random to prevent all of you working on one individual core) by setting hard CPU affinity. Rerun the code, and generate a data file containing the process times. Generate a visualisation that shows when processes are running (e.g. using excel). You should recognise the characteristics of the Linux scheduler that were discussed during the lectures. As always, make sure that the parent process waits for the child processes to finish.

Process Priorities

Background

As a student user (but also for most members of staff) on the school's servers, there are fairly few ways in which you can manipulate processes and process priorities. E.g., you are not able to specify higher than usual priorities for your processes, nor can you change the type of process from the "normal class" to, e.g., the real time class. However, to investigate the influence of process priority on scheduling, it is sufficient to create different process priorities by lowering the priority of some of them. Lowering process priorities is a "one-way street": once you have reduced the priority of one of your processes, you cannot increase it again at a later point in time. Even the child processes inherit the parent's priority values. The restrictions on setting process priorities prevents regular users from creating CPU-hogging processes that overtake all the existing ones. This can be achieved by the setpriority() system call, as illustrated below. More information on the setpriority() call can be found here: link

```
#include <sys/resource.h>

int main(){
    // Sets the priority of the current process
    setpriority(PRIO_PROCESS, getpid(), 19);
}
```

Task 11

Using your code from task 10 (with all processes running on a single CPU), modify the process priorities to be 0, +5, +10, and +15 (i.e. 4 processes). Re-run your code and analyse the results. Generate a data file and visualisation similar to the one above and try to explain the results using the principles discussed during the lectures.