POSIX - Concurrent execution

Introduction

In modern operating systems, the code that runs in the CPU can either be running in a special privilege/protected mode (normally reserved for OS system calls or device drivers) or in the so-called user-space, where user programs execute within **processes**, which are self-contained components with a proper interface with the OS. The OS supports creating, scheduling, terminating processes, and enables the communication/signaling with and between the processes.
Curiosity: The *printf* function manages the interface between a process and a special serial communication channel (aka standard output). When launching a process this channel is bound to the terminal/console, printing the serialized data.

If you develop a monolithic (not concurrent) piece of code, you are good with running it in a single process or thread. However, if you are waiting for an external event (e.g. using *scanf*) that triggers a time-intensive task, one has to choose between blocking on the external event (*wait on scanf*) or executing the task and risk missing input events. In the following weeks we will cover the handling of concurrent task execution in your program with both multi-processing and multi-threading. We will also cover the synchronization of these tasks, sharing memory and handling race-conditions when accessing shared resources (mutual exclusion).

This laboratory work will be divided into 3 parts with the following objectives:
Part A – *OS processes.*
Part B – *Process threads.*
Part C – *Condition variables and other synchronization methods.*
At the end there will be an **individual assessment** through the development of a small program.

Complementary readings:

- Processes API (OSTEP Ch. 5)
- Threads API (OSTEP Ch. 27 -until Sec. 27.3)
- Locked Data Structures (OSTEP Ch. 29)

Part A – OS Processes

A Process within the Operating System is an instance of a running program that includes the respective code and all the associated memory. In a multi-process OS, a process runs along with other processes, competing for CPU time slots, as well as for other hardware resources, especially I/O.

One approach (there are others) to programmatically launch a new process is to **clone** a running process that we will call the **parent process**. This is achieved using the POSIX function **`fork`**. The clone process is called the **child process**. All the parent process memory is replicated (using *memcpy*) and the child process becomes autonomous, i.e., both parent and child processes are managed independently by the OS and its scheduling.

Use the ***`top`*** command to identify the currently running processes, along with their CPU usage share. The program also measures the execution time of each process from two angles: 1) the system clock that does not stop and 2) the process clock that counts the time the process is in the running state.
The following code instantiates a number of processes – there is a defined macro to control the number of processes.
Tweak **HOW_MANY_FORKS** and try to guess the number of CPU cores on your machine/VM. Mind that Intel CPUs with Hyper-threading (SMT) enabled virtually duplicate the number of cores. In a Unix-like OS (Linux, MacOS, BSD, Solaris), the ***`top`*** command (terminal) shows a rank of the running processes, normally exhibiting the top CPU demanding processes, along with their PID (process ID) and name.

Tutorial Program Atp1 (full code @Atp1.c)

```c
#define HOW_MANY_FORKS 2 //Number of forks. Testing from 1 to 4 should be enough

int main() {
    printf("Starting with process id: %d\n", getpid());

    for (int i=0; i<HOW_MANY_FORKS; i++)
        fork();

    printf("Running process id: %d\n", getpid());

#ifdef ENABLE_HEAVY_LOAD
    unsigned long process_ts_start = _timestamp_sec(CLOCK_PROCESS_CPUTIME_ID);
    unsigned long system_ts_start = _timestamp_sec(CLOCK_REALTIME);

    heavy_load();

    printf("Done with pid: %d in (%lu process seconds) and (%lu system seconds).\n",
        getpid(),
        _timestamp_sec(CLOCK_PROCESS_CPUTIME_ID)-process_ts_start,
        _timestamp_sec(CLOCK_REALTIME)-system_ts_start
    );
#endif

    return 0;
}
```

Exercise 1: Modify the function *heavy_load*, so it is busy-waits for a certain amount of time (fixed), say 15 seconds, counted using the CPU process clock. The process clock differs from the real-time clock, in which it only counts time when the process is executing.

The use of process forking involves creating a new memory space – the process memory is in fact duplicated upon forking (memcpy). Hence, it is not possible to share a variable in memory amongst different processes even when cloned from the same program. Memory isolation is enforced by design at OS level.

The following snippet (Atp2) illustrates memory isolation. It also uses the function ***`wait`*** to synchronize the parent process at the end of the child. Examine the code and try to predict the results on each ***printf***.

Tutorial Program Atp2 (full code @Atp2.c)

```
int some_global_var = 0;

int main()
{
    int some_local_var = 0;
    int ret;
    if ( (ret=fork()) == 0){
        // child process running here because return fork returned zero
        printf("Child(%d) says: Hello!\n", getpid());

        printf("Child(%d) says: Global (%p) incremented to %d!\n", getpid(),
               &some_global_var, ++some_global_var );
        printf("Child(%d) says: Local (%p) incremented to %d!\n", getpid(),
               &some_local_var, ++some_local_var );

        printf("Child(%d) says: Done!\n", getpid());

        exit(0); //quits the process immediatly from any point in the program
    }
    else{
        // parent process running here because fork returned value non-zero.
        printf("Parent says: My child's id: %d!\n", ret);

        //Wait until one child exits. NULL means I don't care about the child's exit code.
        int c = wait(NULL);
        printf("Parent says: Child %d is done and so am I!\n", c);

        printf("Parent says: Global (%p) is %d!\n", &some_global_var, some_global_var);
        printf("Parent says: Local (%p) is %d!\n", &some_local_var, some_local_var);
    }

    return 0;
}
```

Exercise 2: Modify the previous code so the parent launches two different Childs and waits (synchronizes) for them before returning.

Multitasking

There are two main arguments that encourage multitasking a process and splitting the execution of a monolithic program: 1) Having multiple execution contexts enables blocking routines (e.g. waiting on a *scanf* or some I/O message) to coexist without blocking each other; and 2) speed up the overall execution by exploring multicore infrastructures and code parallelization.
In both scenarios, there will be a moment when processes need to synchronize and communicate data from one side to the other. Inter-process communication (IPC) is a vast topic that ranges from simple process signaling to complex network-oriented communication and component distribution models. A list of IPC techniques can be found here.
A simple mechanism to retrieve data from a child process to the parent process uses the exit status code. When quitting processes, the program may report a signal code to the OS. This signal normally briefs the OS with an error code that indicates the cause conveying the end of the process. In C language the programmer may use the **`return`** statement in the **`main`** function or the **`exit`** function with the proper code (exit – issues program termination directly from any function).
Curiosity: The following shell command “gcc -o code code.c && ./code” runs a program after a successful compilation process, i.e executes if *gcc* returns code 0 (no error).

In Atp3.c you may find a monolithic code example that executes a batch of jobs, compiles the result for each job and prints out the result at the end. Analyze the code to understand the use of the struct handler that follows the whole process. The structure stores the input values, the pending status and the result.

Exercise 3: Implement the function **`dispatch_jobs_v1`**. This alternate function shall produce the same result, but instead of a monolithic approach, each job runs on a dedicated process. Carry out your tests along with a *top* observation.

Exercise 4: The previous approach has one catch! In a scenario with 1000 jobs you’ll trigger 1000 processes, all competing for the CPU.
Implement the function **`dispatch_jobs_v2`**, similarly to v1, but instead of launching all processes at once, it limits the number of simultaneous processes (workers) – see SIMULTANEOUS_WORKERS_MAX. Try to avoid empty slots – example: If you trigger A and B processes, A faster than B, and you synchronize both before launching C and D, then you’ll be wasting computing power.

Part B – Process Threads

Process threads can be seen as sections of a program that execute independently within the scope of a process. Threads execution may differ, based on the OS architecture, but in general they are components of the process, sharing execution code data and memory (particularly the process data segment and heap).
The following snippet (Btp1) depicts the instantiation of two threads with the same code base, followed by a synchronization at the end of each thread.

Tutorial Program Btp1 (full code @Btp1.c)

```
// A regular C function that executs as a thread
void *myThreadFunction(void *vargp)
{
    int id = *(int *)vargp;
    sleep(2);
    printf("Printing from Thread %d \n", id );
    return NULL;
}

int main()
{
    pthread_t tid_1, tid_2; //Thread ID

    printf("Creating 2 Threads\n");

    pthread_create(&tid_1, NULL, myThreadFunction, (void *)(&tid_1) );
    pthread_create(&tid_2, NULL, myThreadFunction, (void *)(&tid_2) );

    pthread_join(tid_1, NULL);
    pthread_join(tid_2, NULL);

    printf("After Running Threads\n");
    exit(0);
}
```

One of the most relevant aspects that differentiate thread-based and process-based approaches for multitasking is memory access. We have seen with processes that communication between processes requires additional OS components. When using threads, their process memory space is shared, which facilitates data sharing. Though, keep in mind that open access to memory comes with the risk of failure/security and specially data integrity. Memory isolation and protection is valued for certain applications.

The following snippet (Btp2) highlights how threads can seamlessly update global and static variables.
Note: static variables behave as permanent variables within the scope of a function. Static and global variables sit on the data segment of the process. In this example the static variable **s** refers to the same memory position in both threads. You may see static variables as being persistent across multiple calls of a function. In the case of a thread's execution body, it is the same variable across different threads, whereas local variables will still be given different memory positions on each thread's stack.

Tutorial Program Btp2 (full code @Btp2.c)

```c
// Create a global variable to to observe its changes
int g = 0;

// A regular C function that executs as a thread
void *myThreadFunction(void *vargp)
{
    int id = *(int *)vargp;

    // Create a static variable to observe its changes
    static int s = 0;

    // Change static and global variables
    s++; g++;

    //Change again and print
    printf("Thread ID(%p): %d, Static(%p): %d, Global(%p): %d\n",
                  &id, id,        &s, ++s,        &g, ++g);

    return NULL;
}

int main()
{
    pthread_t tid[3]; //Thread ID

    for (int i = 0; i < 3; i++)
        pthread_create(&tid[i], NULL, myThreadFunction, (void *)&tid[i]);

    for (int i = 0; i < 3; i++)
        pthread_join(tid[i], NULL);

    exit(0);
}
```

Memory isolation between processes means that each process gets its private memory space. Forking a process requires an operation of allocating memory for that process (managed at OS level), cloning the current process memory, and binding to several system resources. And all this takes time. It may seem fast to a naked eye, but sure is not at computing level.
The following snippet (Btp3) proposes the execution of an empty task (start and finish) multiple times, running with processes and threads for comparison. You may witness the performance penalty of creating and destroying a process.

Tutorial Program Btp3 (full code @Btp3.c)

```
#define TASKS_NUM 100000

printf("Running %d dummy processes...\n", TASKS_NUM);

clock_gettime(CLOCK_REALTIME, &start);

for (int i=0; i<TASKS_NUM; i++){
    if ( (ret=fork()) == 0){
        // CHILD!
        myThreadFunction(NULL);
        exit(0);
    }else{
        //The PARENT
        wait(NULL); //Wait on the process
    }
}

clock_gettime(CLOCK_REALTIME, &end);

printf("Handled %d Processes in %ld ms!\n\n",
       TASKS_NUM, (end.tv_sec-start.tv_sec)*1000 + (end.tv_nsec-start.tv_nsec)/1000000 );

printf("Running %d dummy threads...\n", TASKS_NUM);

clock_gettime(CLOCK_REALTIME, &start);

for (int i=0; i<TASKS_NUM; i++){
    pthread_create(&tid, NULL, myThreadFunction, NULL );
    pthread_join(tid, NULL);
}

clock_gettime(CLOCK_REALTIME, &end);

printf("Handled %d Threads in %ld ms!\n\n",
       TASKS_NUM, (end.tv_sec-start.tv_sec)*1000 + (end.tv_nsec-start.tv_nsec)/1000000 );
```

Mutexes

In scenarios where multiple threads access shared data or common resources simultaneously it is common to enclose the access within a critical section. In this section we need to guarantee that any operation executes without preemption, i.e no other thread will access the region while one thread is performing. Mutexes behave as binary flags that when locked prevent other threads from entering the associated critical section. *Lock* and *unlock* operations are atomic, guaranteed at kernel or processor level.

For certain data types and usage models there are programming techniques that circumvent the requirement for a strict mutex to enter the critical region. This often requires a deep understanding of the concurrent model, the OS scheduling behavior and compiler instructions regarding atomicity. A simple example would be a thread that only reads data inside that region, which may be dispensed of checking the mutex in certain conditions. In fact, this is not a golden rule though. The use of mutexes or other atomic structures such as semaphores guarantee a safe access to critical sections.

Here's a list of basic primitives to handle a mutex in a multithreading environment:

pthread_mutex_init pthread_mutex_lock pthread_mutex_unlock pthread_mutex_destroy

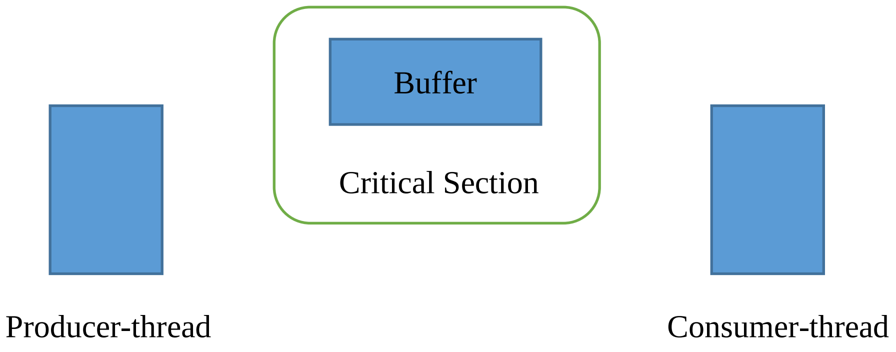
The following snippet (Btp4) instantiates three threads that concurrently print data to the terminal on specific lines. The result is a mess because the access is unguarded.

Tutorial Program Btp4 (full code @Btp4.c)

```
void* printer(void *arg) {

    //Pick a pseudo-random line [1..10]
    int line = (long)pthread_self()%10 + 1;

    while(1) {
        terminal_print_at(line, arg);
        usleep(100*1000); //100 ms
    }

    return NULL;
}


int main(void)
{
    pthread_t tid[3];

    terminal_clear();

    pthread_create(&(tid[0]), NULL, &printer, "I'm a very long long long long long long long long
long long long long long long string");
    pthread_create(&(tid[1]), NULL, &printer, "short string 1");
    pthread_create(&(tid[2]), NULL, &printer, "short string 2");

}
```

Exercise 5: Improve the code in Btp4 to guard the access to the shared resource - the terminal.

We have seen an example of unguarded access to a shared resource. The following snippet (Btp5) depicts the use of two threads in a producer-consumer model over a shared data model – a circular FIFO buffer. There is one thread that periodically pushes values to a circular FIFO buffer, while another thread periodically pops values from that buffer. The producer thread is running faster (smaller period) than the consumer, leading to occasional (but certain) “Buffer Full” messages. You may ignore these messages for now.
With two threads accessing a critical region (the buffer), the buffering structure may become inconsistent. This race-condition requires mutual-exclusion protection (mutexes).



Tutorial Program Btp5 (full code @Btp5.c)

```
void* producer(void *arg)
{
    unsigned int push_value;
    while (1) {
        push_value = (rand() % 1000); //random [0,999]
        if (circ_buff_push(&buffer, push_value ) == 0)
            printf("Producer: %u\n", push_value);
        else
            printf("Producer: buffer is full\n");

        usleep(100*1000); //100 ms
    }

    return NULL;
}

void* consumer(void *arg)
{
    unsigned int pop_value;
    while (1) {
        if (circ_buff_pop(&buffer, &pop_value)==0)
            printf("                                      Consumer: returned %u\n", pop_value);
        else
            printf("                                      Consumer: buffer is empty\n");

        usleep(150*1000); //150 ms
    }

    return NULL;
}
```

Exercise 6: Improve the code in Btp5 to protect the calls to the buffer with a mutex.

Deadlocks

The use of multiple mutexes comes with a risk. When two threads are waiting (to lock) on different mutexes and the unlock of each is inside the critical region of the other mutex, then they enter a state of deadlock, where none can unlock and move on. Btp6 depicts one such scenario.

Tutorial Program Btp6 (full code @Btp6.c)

```c
int counter;
pthread_mutex_t counter_lock;

int mod_value;
pthread_mutex_t mod_value_lock;

void* increment(void *arg) {
    while(1){
        pthread_mutex_lock(&counter_lock);
        counter ++;

        if (counter%10 == 0){
            pthread_mutex_lock(&mod_value_lock);
            mod_value ++;
            usleep(30*1000); //Simulate some heavy computation
            pthread_mutex_unlock(&mod_value_lock);
        }

        pthread_mutex_unlock(&counter_lock);

        printf("\n Increment counter:%d   mod_value:%d\n", counter, mod_value);

        usleep(50*1000); //50 ms
    }
}

void* fastincrement(void *arg) {
    while(1){
        pthread_mutex_lock(&mod_value_lock);
        pthread_mutex_lock(&counter_lock);
        counter ++;
        mod_value += 10;
        //usleep(300*1000); //Simulate some heavy computation
        pthread_mutex_unlock(&counter_lock);
        pthread_mutex_unlock(&mod_value_lock);s
    }
}
```

Exercise 7: Improve the code in Btp6 to avoid deadlocks.

Part C –Threads Synchronization

In Btp5 we have seen two threads making use of the Producer-Consumer model. The producer updates the buffer and the consumer should get the value in the buffer after the update. But both threads are unsynchronized and the buffer handling on both sides constitutes a critical section. The consumer is unaware of when the producer updates the buffer and it keeps polling the queue size to get new data.

This communication/synchronization model is inherently inefficient - the Consumer thread will either busy-wait (CPU intensive loop) constantly checking for updates, or it will only poll from time to time, lowering the CPU demand and potentially increasing the latency between the producing event and the consuming event. To improve this situation, a synchronization mechanism is needed to let the consumer know when there is fresh data in the buffer.

Conditional Variables

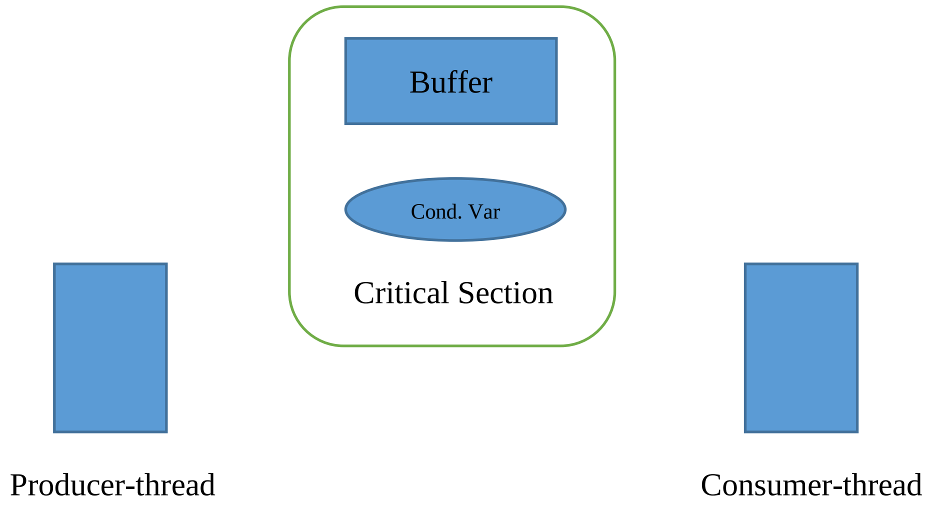
We have seen that mutexes synchronize threads when accessing a critical region. In this case, if the Consumer-thread keeps a mutex locked while waiting for an update, the producer will not be able to push an update because the mutex is locked. Mutexes cannot be locked in one thread and unlocked in another one! Each thread that uses a mutex must lock and unlock it, and only when using the respective critical section. Thus, we need another solution.

Conditional Variables provide a solution for this thread synchronization problem. One can see a conditional variable as a flag inside the critical section. The Consumer thread locks the mutex of the critical section and waits on that flag (in this case, for that flag to be set by the Producer). While waiting, it must temporarily release the mutex, so that whoever sets the flag (the Producer in this case) can enter the critical section, perform its function (updating the buffer in this case) and signal the condition variable.

Once the Producer signals the conditional variable, the Consumer thread immediately[1] leaves the waiting state and locks the given mutex again, knowing that the Producer left the critical section after an update. The consumer is now inside the critical section and ready to fetch the new data. This releasing and reacquiring the mutex while waiting is done automatically inside the respective system call (see the next section).

Conditional variables inside critical sections behave as notification mechanisms (like signaling channels).

[1] Not exactly "immediately" because the producer will have to leave the lock after flagging the update.



Libpthread API

- `pthread_cond_t c = PTHREAD_COND_INITIALIZER;`
  `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
  - Macros with generic initialization code.

- **pthread_cond_wait (pthread_cond_t *restrict *cond*,
  pthread_mutex_t *restrict *mutex*)**
  - A thread <u>must</u> hold the mutex, when calling pthread_cond_wait().
  - pthread_cond_wait() automatically releases the lock while waiting.
  - When the condition variable cond is signaled, pthread_cond_wait() automatically reacquires the lock so that the thread is holding it upon returning. **However,** it is not guaranteed that the condition associated to the signaling, i.e., the flag, is still satisfied – other threads may have preempted this thread after return and taken an action that resets the flag.

- **pthread_cond_signal(pthread_cond_t *cond)**
  - Wakes up one thread waiting on the condition variable, <u>if any</u>.

- **pthread_cond_broadcast(pthread_cond_t *cond)**
  - Wakes up all threads waiting on the condition variable, <u>if any</u>.

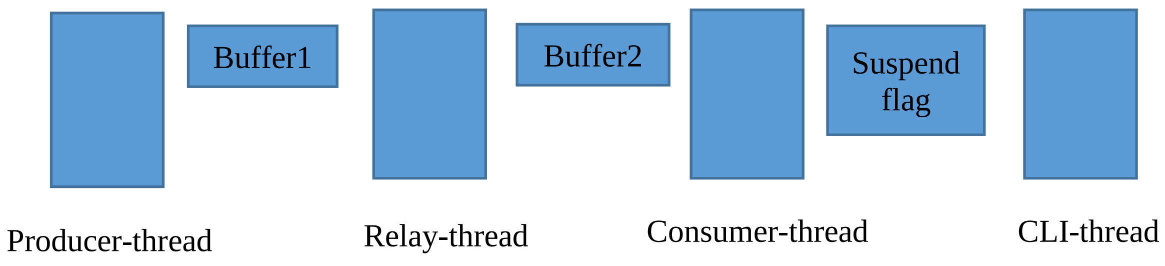
Exercise 8: In Btp5 the access to the shared resource (buffer) was guarded with a mutex. But both threads are running freely in independent loops – unsynchronized. Add a conditional variable to the code, so that the consumer polls the queue status only when notified. Be aware that:

- When the producer signals the conditional variable, the consumer may not be listening (i.e., waiting within pthread_cond_wait()). As an example, if the consumer is notified and then takes too much time processing the buffer until returning to waiting again, the producer may generate one or multiple signal notifications in the meanwhile, that will be overwritten, thus lost.

Exercise 9: Implement a variation of the previous with one producer and two/three consumer threads. Each consumer thread will pop from the buffer and print the result (along with its thread ID).

- Observe how the consumer threads alternate, indicating the existence of a thread waiting queue in the conditional variable.
- Repeat with a broadcast signaling.

Exercise 10: Implement the following scenario, as depicted in the following illustration.



- **Producer-thread** periodically pushes a value (random) to Buffer1.
- **Relay-thread** consumes from Buffer1 and produces to Buffer2. Use a conditional var to sync with the production events.
- **Consumer thread** consumes from Buffer 2 (in sync with Relay-thread production) and prints the output to the screen.
- **CLI-thread** reads from stdin to toggle a flag that suspends the Consumer-thread's activity.
- While suspended, both buffers will get filled up and the Producer-thread eventually gets a full buffer indication. When that happens, the thread should wait for buffer availability. That means the Producer-thread will proceed with its data production loop as soon as Buffer1 becomes available again. This procedure needs to be propagated for Buffer2, so that when Consumer-thread resumes activity, Relay-thread can relay from Buffer 1 to Buffer2, thus waking Producer-thread that can now push data to Buffer1.
- Another point to consider is that Relay-thread must only pop from Buffer1 if it is certain that it fits in Buffer2 (not full).