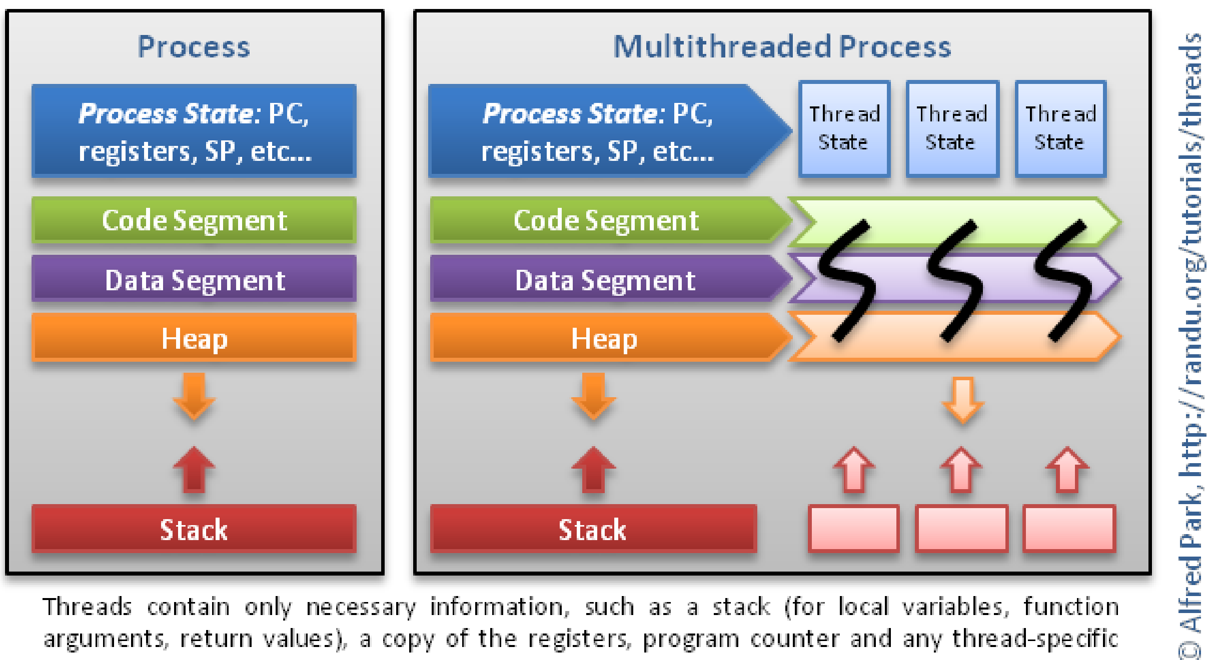
[PB167 /PB167/](#) Exercise #6 Linux threads & inter-process synchronization.

Meet important types & functions:

- thread type: **pthread_t**
- int **pthread_create**(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void *(*start_routine)(void *), void *restrict arg);
- int **pthread_join**(pthread_t thread, void **value_ptr);



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

http://randu.org/tutorials/threads/images/process.png

- Simple thread example:

http://www.csc.villanova.edu/~mdamian/threads/badcnt.txt

- Compile with no optimization -O0 (so we have race condition)
- In order to slow down computation, add printf("."); after cnt modification.

Tasks:

a. Only a single value can be passed to the main thread function, modify it so we can pass a structure.
b. Modify a given program so we can start N threads (given as an argument in argv[]).

   ```
   int numThreads = atoi(argv[1]);
   ```

Mutex

Basic primitive for implementation of a **critical section**.

- pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
- int pthread_mutex_lock(pthread_mutex_t *mutex);
- int pthread_mutex_trylock(pthread_mutex_t *mutex);
- int pthread_mutex_unlock(pthread_mutex_t *mutex);
- int pthread_mutex_destroy(pthread_mutex_t *mutex);

Task:

a. Edit badcnt.txt,
    i. Declare a global integer **volatile** variable startCond, set default value to 0.
    ii. Each thread will wait in a for-loop (busy-waiting) for a startCond to became 1.
    iii. Set startCond to 1 after all threads have been created in the main().
b. Start badcnt.txt with 15 threads and observe given numeric values.
c. Try to fix the program with use of **mutex** on correct places.
    i. You will need one pthread_mutex_t mutex variable.

Semaphore

- Generalized mutex, with counter inside.
- Counter has to be **non-negative**.
- Counter can be atomically **decremented**.
    - If is 0, decrementing threads blocks until counter can be decremented to non-negative value.
- Counter can be atomically **incremented**.
    - Does not block, we can always increment.
    - If a thread X is waiting to decrement counter, inc operation causes X thread wake-up.

- Mutex can be implemented with a semaphore with counter set to 1.
- Lock = decrement.
- Unlock = increment.

Important types & functions:

- sem_t sem_name;
- int sem_init(sem_t *sem, int pshared, unsigned int value);
- int sem_wait(sem_t *sem);
- int sem_post(sem_t *sem);
- int sem_getvalue(sem_t *sem, int *valp);
- int sem_destroy(sem_t *sem);

Task:

- Use semaphore as a signaling primitive.
- Example:
    - We have a working thread which produce some data. E.g., 5 working threads.

  - We have a consumer thread which needs produced data. Consumer thread needs at least 3 results.

- Start 5 worker threads producing data.
  - For loop, sleep 1 second, increment globally shared counter value (protected by mutex).
- Start consumer thread, which writes a line after at least 3 results were produced -- 3 consecutive sem_wait() calls...

Condition variable

Condition variables allow threads to synchronize to a value of a shared resource. Typically, condition variables are used as a notification system between threads.

- Condition variable is tied to a mutex.
- **pthread_cond_t** cond = PTHREAD_COND_INITIALIZER;
- int **pthread_cond_wait**(pthread_cond_t *cond, pthread_mutex_t *mutex);
  - When calling wait, mutex has to be owned by a calling thread. Logic: lock the mutex, check for a condition safely (in a loop), wait for condition trigger.
  - Has to be done in a loop in order to avoid spurious wakeups - verify condition once again after wakeup, just to be sure.
  - When wait is triggered, mutex is again owned by waiting thread so we can check condition safely. (re-acquire).
    - lock the mutex (sanity)
    - while(condition) pthread_cond_wait(cond, mutex);
- int **pthread_cond_signal**(pthread_cond_t *cond);
  - Wakes up one (out of many possible) waiting thread(s).
  - Locks that other threads could be waiting on should be released before you signal or broadcast.
- int **pthread_cond_broadcast**(pthread_cond_t *cond);
  - Wakes up all waiting threads on this condition variable.

```c
void *thr_func1(void *arg) {
  /* thread code blocks here until MAX_COUNT is reached */
  pthread_mutex_lock(&count_lock);
    while (count < MAX_COUNT) {
      pthread_cond_wait(&count_cond, &count_lock);
    }
  pthread_mutex_unlock(&count_lock);
  /* proceed with thread execution */

  pthread_exit(NULL);
}

/* some other thread code that signals a waiting thread that MAX_COUNT has been reached */
void *thr_func2(void *arg) {
  pthread_mutex_lock(&count_lock);

  /* some code here that does interesting stuff and modifies count */

  if (count == MAX_COUNT) {
    pthread_mutex_unlock(&count_lock);
    pthread_cond_signal(&count_cond);
  } else {
    pthread_mutex_unlock(&count_lock);
  }

  pthread_exit(NULL);
}
```

Task:

Implement buffer-bound (fixed buffer - resembles PIPE) producer/consumer, with 10 producer and 5 consumer threads. Size of a buffer = 6.

- Buffer type: unsigned long *.
- Producers produce a random odd numbers to a buffer of size 6.
- Consumer will test numbers in the input buffer for primality.
  - Use either naive division up to sqrt(X) or Google up Rabin-Miller primality test.
  - When prime is detected, consumer writes it to a shared file, protected by a mutex.

- Cheat: http://pages.cs.wisc.edu/~remzi/OSTEP/threads-cv.pdf