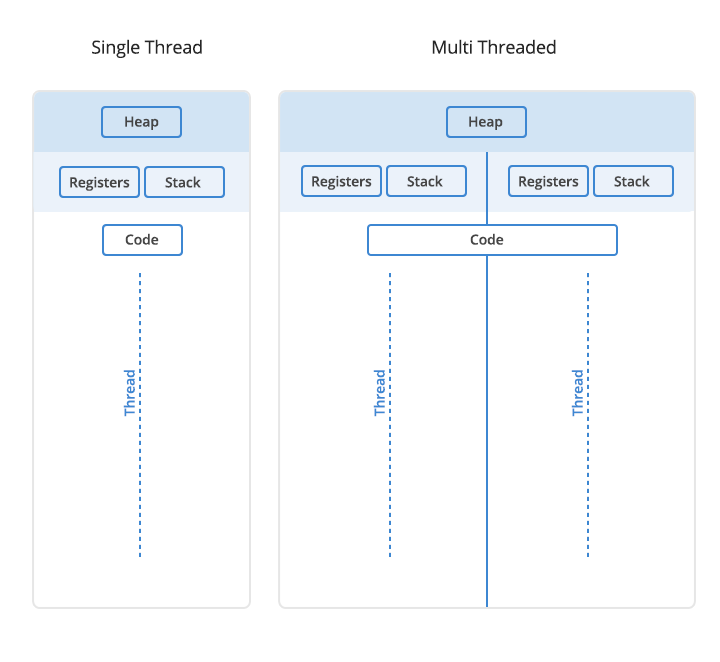
Threads

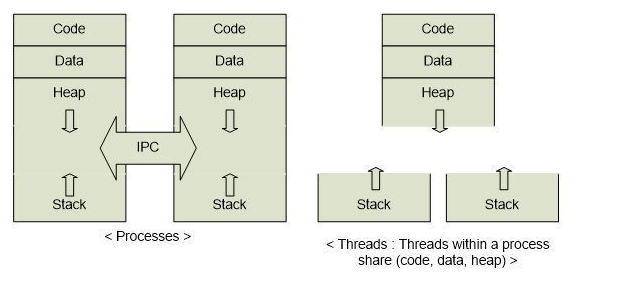
# Thread, Mutex, RWLock, Conditional variable

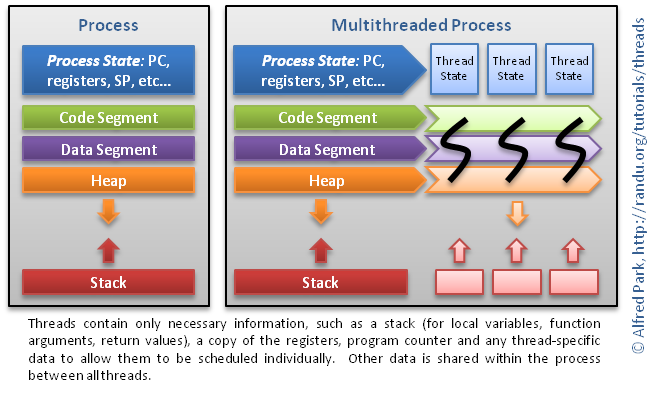
|  |  |
| --- | --- |
| **Header file** | <pthread.h> |
| **Compile using library** | -pthread |
| **Thread function definition** | void \* work(void\* a) |
| **Data types** | pthread\_t  pthread\_mutex\_t  pthread\_cond\_t  pthread\_rwlock\_t  sem\_t |
| **Thread Functions** | int **pthread\_create**(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*arg);  int **pthread\_join**(pthread\_t thread, void \*\*retval);  void **pthread\_exit**(void \*retval); |
| **Mutex Functions** | int **pthread\_mutex\_init**(pthread\_mutex\_t \*restrict mutex,  const pthread\_mutexattr\_t \*restrict attr);  //or pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;  int **pthread\_mutex\_lock**(pthread\_mutex\_t \*mutex);  int **pthread\_mutex\_unlock**(pthread\_mutex\_t \*mutex);  int **pthread\_mutex\_destroy**(pthread\_mutex\_t \*mutex); |
| **Conditional variable functions** | pthread\_cond\_init  pthread\_cond\_wait  pthread\_cond\_signal  pthread\_cond\_broadcast  pthread\_cond\_destroy |
| **R/W Lock Functions** | int **pthread\_rwlock\_init**(pthread\_rwlock\_t \*restrict rwlock,  const pthread\_rwlockattr\_t \*restrict attr);  //or pthread\_rwlock\_t rwlock = PTHREAD\_RWLOCK\_INITIALIZER;  int **pthread\_rwlock\_rdlock**(pthread\_rwlock\_t \*rwlock);  int **pthread\_rwlock\_wrlock**(pthread\_rwlock\_t \*rwlock);  int **pthread\_rwlock\_unlock**(pthread\_rwlock\_t \*rwlock);  int **pthread\_rwlock\_destroy**(pthread\_rwlock\_t \*rwlock); |
| **Semaphore Functions** | sem\_init  sem\_wait  sem\_post  sem\_destroy |

# Threads

** "Threads are just boneless processes." - Mahatma Gandhi (???)**







A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables).

Threads share a range of other attributes: PID, PPID, open file descriptors, locks, signal dispositions, etc. Threads do NOT share (differ in): thread ID, errno variable, scheduling/priority, etc.

1. Write a program that receives strings as command line arguments and uses threads to capitalise each word. We’ll create for each argument a separate thread that will capitalise the initial letter.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAXLINIE 1000  pthread\_t tid[100]; //we need to refer to each thread to join them  void\* ucap(void\* numei) {  printf("Thread start: %ld ...> %s\n", pthread\_self(), (char\*)numei);    char numeo[100];  strcpy(numeo, (char\*)numei);  if ( numeo[0]>=`a` && numeo[0]<=`z`)  numeo[0]+=’A’-‘a’;  printf("Thread finished: %ld > %s\n", pthread\_self(), (char\*)numeo);  }  int main(int argc, char\* argv[]) {  int i;  for (i=1; argv[i]; i++) {  pthread\_create(&tid[i], NULL, ucap, (void\*)argv[i]);  printf("Thread created: %ld ...> %s\n", tid[i], argv[i]);  }  for (i=1; argv[i]; i++) pthread\_join(tid[i], NULL);  printf("All threads finished\n");  } |

Compile: gcc -pthread capit.c, and run ./a.out f1 f2 . . .

If your system is missing manual pages for thread, mutex etc. function you can install it:

sudo apt-get install manpages-posix manpages-posix-dev

sudo apt-get install glibc-doc

**The return value of a thread**

1. Solve the problem of adding 4 numbers in parallel, but using two threads – one that calculates the sum of first two, another that calculated the sum of last two, and main program receives the calculated values and uses the partial sums to compute the total sum. For the return value we use pthread\_exit and pthread\_join (second argument) together with a pointer to a return\_val which can be, as in our case, a struct.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAXLINIE 1000  pthread\_t tid[100]; //we need to refer to each thread to join them  int a[]={1,2,3,4,5,6,7,8,9,10};  typedef struct { int ps; } response;  void\* partial(void\* id) {  int nr= \*(int \*)id;  response \* r= malloc(sizeof(response));  printf("Thread start: %ld ...> %d\n", pthread\_self(), nr);  r->ps=a[2\*nr]+a[2\*nr+1];  printf("Thread finished: %ld > %d\n", pthread\_self(), nr);  pthread\_exit(r);  }  int main(int argc, char\* argv[]) {  response \*r0, \*r1;  int tnr[100];  int i=0;  for (i=0; i<100; i++) {  tnr[i]=i;  }  pthread\_create(&tid[0], NULL, partial, (void\*)&tnr[0]);  pthread\_create(&tid[1], NULL, partial, (void\*)&tnr[1]);  pthread\_join(tid[0], &r0);  pthread\_join(tid[1], &r1);  printf("All threads finished\n");  printf("Total sum is: %d \n", ((response \*)r0)->ps + r1->ps);  free(r0);  free(r1);  return 0;  } |

Homework – generalise the problem above to work for performing the parallel addition of 10 or more numbers n, using n/2 threads that compute the sum of pairs from index 2t and 2t+1, where t is the thread id. Use a separate for each section – to create the threads, to join the threads, to free memory.

# Mutex

Why do we need such a thing? Synchronization when using shared resources and we have **race conditions** and at least a **critical section**. The solution must ensure that:

**1. No two processes may be simultaneously inside their critical regions.**

**2. No assumptions may be made about speeds or the number of CPUs.**

**3. No process running outside its critical region may block other processes.**

**4. No process should have to wait forever to enter its critical region.**

1. Write a program that has a global variable count and 1000 threads. Each thread increments count 1000 times. The following code will output different numbers in each execution.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  int count = 0;  pthread\_t tid[1000];  void\* inc(void\* nume) {  for (int i = 0; i < 1000; i++) {  int temp = count; temp++; count = temp;  }  }  int main(int argc, char\* argv[]) {  int i;  for (i=0; i < 1000; i++)  pthread\_create(&tid[i], NULL, inc, NULL);  for (i=0; i < 1000; i++) pthread\_join(tid[i], NULL);  printf("count=%d\n", count);  } |

What happens, why we have sometimes 1000000, sometimes 991000?

The three instructions inside the **for** of the **inc** thread functions are mixing up with those from the other threads running. What happens is that sometimes a thread reads a value that was already modified and changes it, causing loss of increments performed by other threads. This happens even if we replace

int temp = count; temp++; count = temp;

with count++;

because the count++ in itself is performed by the processor in 3 distinct steps:

1. read/copy the value of count in registers,
2. do the incremental in registers,
3. Copy/move the new value from registers to the address of the variable count.

Any thread can be interrupted in its execution between the three operations, and meanwhile the value of n can change, while the thread will use the old read value when continuing its execution.

To ensure that whenever a thread reads a variable and need to modify it no other thread can read it until the first one finishes, we use a mutex.

We declare a global mutex: pthread\_mutex\_t exclusive= PTHREAD\_MUTEX\_INITIALIZER;

Alternativelu we may initialize it in main: pthread\_mutex\_init(&exclusive, NULL);

Surround the **critical code section** with lock/unlock:

pthread\_mutex\_lock(&exclusive);

int temp = count; temp++; count = temp;

pthread\_mutex\_unlock(&exclusive);

Deallocate at the end:

pthread\_mutex\_destroy(&exclusive);

1. Given n pairs of command line arguments which are integer numbers, computer how many pairs have (a) and even sum, (b) and odd sum, (c) at least one the arguments is 0 or nonnumerical. We will create a thread for each pair, and three global variables for counting the three conditions, which will be accessed exclusively by each thread. We should be using a separate mutex for each – we want to correctly synchronise and have as much as possible performed in parallel.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #define MAXLINIE 1000  typedef struct {char\*n1; char\*n2;} PERECHE;  pthread\_t tid[100];  PERECHE pair[100];  pthread\_mutex\_t mut = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_mutex\_t mtxeven = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_mutex\_t mtxodd = PTHREAD\_MUTEX\_INITIALIZER;  int pare = 0, impare = 0, nenum = 0;  void\* computepairs(void\* pair) {  int n1 = atoi(((PERECHE\*)pair)->n1);  int n2 = atoi(((PERECHE\*)pair)->n2);  if (n1 == 0 || n2 == 0) {  pthread\_mutex\_lock(&mut);  nenum++;  pthread\_mutex\_unlock(&mut);  }  else if ((n1 + n2) % 2 == 0) {  pthread\_mutex\_lock(&mtxeven);  pare++;  pthread\_mutex\_unlock(&mtxeven);  }  else {  pthread\_mutex\_lock(&mtxodd);  impare++;  pthread\_mutex\_unlock(&mtxodd);  }  }  int main(int argc, char\* argv[]) {  int i, p, n = (argc-1)/2;  for (i = 1, p = 0; p < n; i += 2, p++) {  pair[p].n1 = argv[i];  pair[p].n2 = argv[i+1];  pthread\_create(&tid[p], NULL, computepairs, (void\*)&pair[p]);  // We need to allocate separate memory for each thread argument!! // Trying to reuse this variable will mess-up the code and the threads will //not receive their arguments correctly. Incorect to say ..., (void\*)&pair);  }  for (i=0; i < n; i++)  pthread\_join(tid[i], NULL);  printf("pairs=%d even=%d odd=%d nonnumeric=%d\n",n,pare,impare,nenum);  pthread\_mutex\_destroy(&mut);  pthread\_mutex\_destroy(&mtxodd);  pthread\_mutex\_destroy(&mtxeven);  return 0;  } |

1. What happens if we reuse variable pair instead of an array of pairs, one for each thread?
2. What happens if we place create and join in the same for loop?

for (i = 1, p = 0; p < n; i += 2, p++) {

pair[p].n1 = argv[i];

pair[p].n2 = argv[i+1];

pthread\_create(&tid[p], 0, computepairs, (void\*)&pair[p]);

pthread\_join(tid[i], NULL);

}

1. Why we use 3 mutexes instead of 1?

# Read Write Lock – Multiple readers, few writers problem

A read lock pthread\_rwlock\_rdlock allows any number or readers to access the resources for reading only, but no writers are allowed. A write lock pthread\_rwlock\_wrlock allows only one writer to access the resource, no other writers/readers allowed.

It is the classical database access problem – a lot of people read, very few modify. For example online catalogue of grades/attendance – edited by the professor in rare occasions, read often by many students.

1. Write a program to simulate this problem of multiple readers reading symoultaneously and only one writer at a time.

|  |  |
| --- | --- |
| States of a writer thread (S):  -3 writer not started,  -2 writer managed to write and will sleep,  -1 waits for readers to finish their operations,  -0 writer is writing. | States of a reader thread (C):  -3 reader not started yet,  -2 reader managed to read and will sleep,  -1 readers waits for writers to finish,  0 reader is reading. |

|  |
| --- |
| #include <pthread.h>  #include <stdlib.h>  #include <unistd.h>  #include <stdio.h>  #define C 7 //readers  #define S 2 //writers  #define CSLEEP 2  #define SSLEEP 3  pthread\_t tid[C + S]; //some writer threads, some reading threads  int c[C], s[S], nt[C + S];  pthread\_rwlock\_t rwlock;  pthread\_mutex\_t exclusafis;  //print states of readers and writers  void afiseaza() {  int i;  pthread\_mutex\_lock(&exclusafis);  for (i = 0; i < C; i++) printf("C%d\_%d\t",i, c[i]);  for (i = 0; i < S; i++) printf("S%d\_%d\t",i, s[i]);  printf("\n");  fflush(stdout);  pthread\_mutex\_unlock(&exclusafis);  }  //reader thread function  void\* cititor(void\* nrc) {  int indc = \*(int\*)nrc;  for ( ; ; ) {  c[indc] = -1; // Waits to read if locked by a writer  pthread\_rwlock\_rdlock(&rwlock);  c[indc] = 0; // Reads  afiseaza();  sleep(1 + rand() % CSLEEP); //simulate reading operation time  c[indc] = -2; // Reading finished and sleeps  pthread\_rwlock\_unlock(&rwlock);  sleep(1 + rand() % CSLEEP);  }  }  //writer thread function  void\* scriitor (void\* nrs) {  int inds = \*(int\*)nrs;  for ( ; ; ) {  s[inds] = -1; // Waiting to write    pthread\_rwlock\_wrlock(&rwlock);  s[inds] = 0; // Writes  afiseaza();  sleep(1 + rand() % SSLEEP); //similate writing operation time  s[inds] = -2; // Write operation finished and sleeps  pthread\_rwlock\_unlock(&rwlock);  sleep(1 + rand() % SSLEEP);  }  }    int main() {  pthread\_rwlock\_init(&rwlock, NULL);  pthread\_mutex\_init(&exclusafis, NULL);  int i;  for (i = 0; i < C; c[i] = -3, nt[i] = i, i++); // -3 : State of Not started  for (i = 0; i < S; s[i] = -3, nt[i + C] = i, i++);    //launch threads  for (i = 0; i < C; i++) pthread\_create(&tid[i], NULL, cititor, &nt[i]);  for (i = C; i < C + S; i++) pthread\_create(&tid[i], NULL, scriitor, &nt[i]);  for (i = 0; i < C + S; i++) pthread\_join(tid[i], NULL);  pthread\_rwlock\_destroy(&rwlock);  pthread\_mutex\_destroy(&exclusafis);  } |

1. 100 participants in the BEST COMPUTER SCIENCE MEMES contest are waiting for the results. Three sponsors from big companies provide each a prize. Model this problem using threads, in which each competitor checks randomly every few seconds for the announced winners, until they hear their number or all three winners were announced. The jury sponsors take a larger number of seconds to deliberate and state their winners.

|  |
| --- |
| #include <pthread.h>  #include <stdlib.h>  #include <unistd.h>  #include <stdio.h>  #define C 20 //competitors  #define S 3 //sponsors  #define CSLEEP 2  #define SSLEEP 3  pthread\_t tid[C + S]; //some writer threads, some reading threads  int nt[C + S]; // if we wnt to pass ids instead of i  int nrp=0;  int p[S];  pthread\_rwlock\_t rwlock;  //writer thread function  void\* sponsor (void\* i) {  int sponsor = (int)i;  // or \*(int \*) nt if we use the array version nt[] to pass arg thread id    sleep(3 + rand() % SSLEEP); //deliberate  pthread\_rwlock\_wrlock(&rwlock);  p[nrp]=rand()%(C+1);  nrp++;  printf("Winner is: %d \n", p[nrp-1]);  pthread\_rwlock\_unlock(&rwlock);  }  //reader thread function  void\* competitor(void\* nrc) {  int ct = (int)nrc;  int f=0, i=0;  while (f==0 && nrp<S){  pthread\_rwlock\_rdlock(&rwlock);  printf("%d is cheking the winners \n", ct);    for ( i=0; i<nrp; i++){  if (p[i]==ct) {  f=1; printf("Winner me %d!!! \n", ct);  }  }  pthread\_rwlock\_unlock(&rwlock);  sleep(1 + rand() % CSLEEP);  }  //do a last check for the last winner announced  pthread\_rwlock\_rdlock(&rwlock)  for ( i=0; i<nrp; i++){  if (p[i]==ct) {  f=1; printf("Winner me %d!!! \n", ct);  }  }  pthread\_rwlock\_unlock(&rwlock);  }    int main() {  pthread\_rwlock\_init(&rwlock, NULL);  pthread\_mutex\_init(&exclusafis, NULL);  int i;  // for (i = 0; i < S + C; nt[i] = i, i++);  //launch threads – i warding integer to pointer cast; use instead &nt[i]  for (i = 0; i < C; i++) pthread\_create(&tid[i], NULL, competitor, i);  for (i = C; i < C + S; i++) pthread\_create(&tid[i], NULL, sponsor, i);  for (i = 0; i < C + S; i++) pthread\_join(tid[i], NULL);  pthread\_rwlock\_destroy(&rwlock);  pthread\_mutex\_destroy(&exclusafis);  return 0;  } |

1. Similar problem could be the case of Worldometer statistics website, people loading the page to see the number of COVID-19 cases (readers) and authorities updating with new cases (writers) from time to time. Similar to example 5, readers and writers are operating in a continuous loop with random sleeps between reads and writes. Model and implement this problem using threads and RWlocks to maximise efficiency.
2. Try the implementation without using external sources (only terminal and manual).
3. Experiment with different number of threads for read/write and various intervals for the random sleeps. Too many readers may not leave resources for writers too often… see what happens.
4. Replace the RWlock with Mutex and observe the difference over a period of time.

# Hardcore training with threads

Add in parallen n numbers using threads, given a global array *a[0], a[1], . . . a[n-1].*

Idea: create a structure tree hierarchy of threads that wait for other threads to compute partial sums. For example for n=32 we need 31 threads. For example thread 16 calculates the sum of a[0] and a[1], and so on, thread 31 computeas the sum of a[30]+a[31]. The next layer of threads use these partial sums, for example thread 8 computes the partial sum from threads 16 and 17, so the sum of the first 4 numbers.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  int n, m; // n = numarul de operanzi; m = min {2^k >= n}  int\* a; // valoarea 1 pentru pana la n-1, 0 de la n la m-1  pthread\_t \*tid; // id-urile threadurilor; -1 thread nepornit  pthread\_mutex\_t print = PTHREAD\_MUTEX\_INITIALIZER; // Printare exclusiva  // Rutina thread-ului nr i de adunare  void\* aduna(void\* pi) {  int i, j, sa, da, st = 0, dr = 0, k;  i = \*(int\*)pi; // Retine numarul threadului  if (i < m / 2) {  st = 2 \* i; // Retine fiul stang  dr = st + 1; // Retine fiul drept  while (tid[st] == -1); // Asteapta sa inceapa fiul stang  // while (tid[st] == -1) sleep(1); // poate asa!  // Cel mai sanatos este să se utilizeze un set de variabile coditionale  // care sa semnaleze pornirile threadurilor.  while (tid[dr] == -1); // Asteapta sa inceapa fiul drept  // while (tid[dr] == -1) sleep(1); // poate asa!  pthread\_join(tid[st], NULL); // Asteapta sa se termine fiul stang  pthread\_join(tid[dr], NULL); // Asteapta sa se termine fiul drept  }  for (j = m; j > i; j /= 2); // Determina fratele cel mic  for (k = j, sa = 0; k < i; k++) sa += m / j; // operand stang  da = sa + m / j / 2; // operand drept  a[sa] += a[da]; // Face adunarea proppriu-zisa  pthread\_mutex\_lock(&print);// Asigura printare exclusiva  printf("Thread %d: a[%d] += a[%d]", i, sa, da);  if (st > 0) printf(" (dupa fii %d %d)\n", st, dr); else printf("\n");  pthread\_mutex\_unlock(&print);  }  // Functia main, in care se creeaza si lanseaza thread-urile  int main(int argc, char\* argv[]) {  n = atoi(argv[1]); // Numarul de numere de adunat  for (m = 1; n > m; m \*= 2); // m = min {2^k >= n}  int\* pi;  int i;  a = (int\*) malloc(m\*sizeof(int)); // Spatiu pentru intregii de adunat  pi = (int\*) malloc(m\*sizeof(int)); // Spatiu pentru indicii threadurilor  tid = (pthread\_t\*) malloc(m\*sizeof(pthread\_t)); // id-threads  for (i = 0; i < n; i++) a[i] = 1; // Aduna numarul 1 de n ori  for (i = n; i < m; i++) a[i] = 0; // Completeaza cu 0 pana la m  for (i = 1; i < m; i++) tid[i] =-1; // Threadurile sunt inca nepornite  for (i = 1; i < m; i++) pi[i] = i; // Threadurile sunt inca nepornite  for (i = 1; i < m; i++)  // De ce folosim mai jos &pi[i] in loc de &i? vezi un exemplu precedent!  pthread\_create(&tid[i], NULL, aduna, (void\*)(&pi[i])); // Threadul i  pthread\_join(tid[1], NULL); // Asteapta dupa primul thread  printf("Terminat adunarile pentru n = %d. Total: %d\n", n, a[0]);  free(a); // Eliberaza tabloul de numere  free(pi); // Elibereaza tabloul de indici de threaduri  free(tid); // Elibereaza tabloul de id-uri de threaduri  } |

# Conditional variables

Considering two threads, one waits for an even to happen to be able to continue. We call it **rendezvous**.

|  |  |
| --- | --- |
| B | A |
| void\* produceevent(void\* nume) {  ...  printf("B producing event\n");  event = 1;  printf("Event done\n");  ...  } | void\* waitevent(void\* nume) {  ...  printf("A waiting event\n");  while (event == 0) {  ;  }  printf("A received event\n");  ...  } |

The problem is that while performs an **active waiting (busy waiting)**, the processor is occupied by the tread with running the while instruction. To make use this wasted time, we use a conditional variable with an associated mutex:

pthread\_cond\_t var = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t varmtx = PTHREAD\_MUTEX\_INITIALIZER;

Code changes into:

|  |  |
| --- | --- |
| pthread\_mutex\_lock(&varmtx);  event = 1;  pthread\_cond\_signal(&var);  pthread\_mutex\_unlock(&varmtx); | pthread\_mutex\_lock(&varmtx);  while (event == 0) {  pthread\_cond\_wait(&var,&varmtx);  }  pthread\_mutex\_unlock(&varmtx); |

Pthread\_cont\_wait will:

1. Unlock the mutex
2. Wait for a signal
3. When signal comes, lock mutex and continue execution

Pthred\_cond\_signal will:

1. Trigger a signal that will wake wait

# Semaphores

The POSIX system in Linux presents its own built-in semaphore library. To use it, we have to:

1. #include <semaphore.h>
2. Compile the code by linking with -lpthread -lrt

To lock a semaphore or wait we can use the **sem\_wait** function:

int sem\_wait(sem\_t \*sem);

To release or signal a semaphore, we use the **sem\_post** function:

int sem\_post(sem\_t \*sem);

A semaphore is initialised by using **sem\_init**(for processes or threads) or **sem\_open** (for IPC).

sem\_init(sem\_t \*sem, int pshared, unsigned int value);

Where,

* 1. **sem**: Specifies the semaphore to be initialized.
  2. **pshared**: This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.
  3. **value**: Specifies the value to assign to the newly initialized semaphore.

To destroy a semaphore, we can use **sem\_destroy**.

sem\_destoy(sem\_t \*mutex);

**Example: A binary semaphore has the effect of a mutex. The same is the case of a write lock.**

#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

sem\_t mutex;

pthread\_t t1[10];

void\* thread(void\* arg) {

    sem\_wait(&mutex); //mutex lock

    //critical section

    sem\_post(&mutex); //mutex unlock

}

int main()

{

    sem\_init(&mutex, 0, 1); //initialize semaphore as binary

for ... pthread\_create(&t1[i],NULL,thread,NULL);

    for ... pthread\_join(t1[i],NULL);

    sem\_destroy(&mutex);

    return 0;

}

# Barrier

#include <pthread.h>

int **pthread\_barrier\_destroy**(pthread\_barrier\_t \*barrier);

int **pthread\_barrier\_init**(pthread\_barrier\_t \*restrict barrier, const pthread\_barrierattr\_t \*restrict attr, unsigned count);

int **pthread\_barrier\_wait**(pthread\_barrier\_t \*barrier);

The calling thread shall block until the required number of threads have called pthread\_barrier\_wait() specifying the barrier.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <time.h>

#define THREAD\_COUNT 4

pthread\_barrier\_t mybarrier;

void\* threadFn(void \*id\_ptr) {

int thread\_id = \*(int\*)id\_ptr;

int wait\_sec = 1 + rand() % 5;

printf("thread %d: Wait for %d seconds.\n", thread\_id, wait\_sec);

sleep(wait\_sec);

printf("thread %d: I'm ready...\n", thread\_id);

pthread\_barrier\_wait(&mybarrier);

printf("thread %d: going!\n", thread\_id);

return NULL;

}

int main() {

int i;

pthread\_t ids[THREAD\_COUNT];

int short\_ids[THREAD\_COUNT];

srand(time(NULL));

pthread\_barrier\_init(&mybarrier, NULL, THREAD\_COUNT + 1);

for (i=0; i < THREAD\_COUNT; i++) {

short\_ids[i] = i;

pthread\_create(&ids[i], NULL, threadFn, &short\_ids[i]);

}

printf("main() is ready.\n");

pthread\_barrier\_wait(&mybarrier);

printf("main() is going!\n");

for (i=0; i < THREAD\_COUNT; i++) {

pthread\_join(ids[i], NULL);

}

pthread\_barrier\_destroy(&mybarrier);

return 0;

}

W hen the required number of threads have called pthread\_barrier\_wait() specifying the barrier, the constant

THREAD\_BARRIER\_ SERIAL\_THREAD shall be returned to one unspecified thread and zero shall be returned to each of the remaining threads. At this point, the barrier shall be reset to the state it had as a result of the most recent pthread\_barrier\_init() function that referenced it.

# Problems & Exercises

### Threads

1. What will be printed by the code below?

**R:** On each line, in this order:

Thread ID – 0

Thread ID – 1

....

Thread ID – 100

Finished

1. What needs to be changed in the code so that the threads with run concurrently?

**R:**

for (i=0; i<100; i++)

pthread\_create(&tid[0], NULL, partial, (void\*)&tnr[0]);

for (i=0; i<100; i++)

pthread\_join(tid[0], NULL);

1. In case B, what will be printed at different executions?

**R:** The lines from A, first 100 lines in any order, at the end the line with Finished.

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAXLINIE 1000  pthread\_t tid[100];  void\* partial(void\* id) {  int nr= \*(int \*)id;  printf("Thread %ld - %d\n", pthread\_self(), nr);  }  int main(int argc, char\* argv[]) {  int tnr[100];  int i=0;  for (i=0; i<100; i++) {  tnr[i]=i;  }  for (i=0; i<100; i++) {  pthread\_create(&tid[i], NULL, partial, (void\*)&tnr[i]);  pthread\_join(tid[i], NULL);  }  printf("Finished\n");  return 0;  } |

1. What will the following code print?

**R:** Any number between 1 and 7.

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  int n=0;  void\* f(void \* a){  n++;  return NULL;  }  int main(){  int i;  pthread\_t t[7];    for (i=0; i<7; i++)  pthread\_create(&t[i], NULL, f, NULL);  for (i=0; i<7; i++)  pthread\_join(t[i], NULL);  printf("%d \n", n);  return 0;  } |

1. What will print the code below if we run the program with parameters 1 2 3 4?

**R:** It may print 1 2 3 4 or 1 2 1 2 or 3 4 3 4 or any other combination because we keep the same address for a pair for both threads, but the value of the structure change in time, depending how and when threads will execute their code.

1. Modify the program to print 1 2 3 4 or 3 4 1 2.

**R:** we declarePERECHE pair[100]; And modify the for sequence:

for (i = 1, p = 0; p < n; i += 2, p++) {

pair[p].n1 = argv[i];

pair[p].n2 = argv[i+1];

pthread\_create(&tid[p], NULL, computepairs, (void\*)&pair[p]);

}

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  typedef struct {char\*n1; char\*n2;} PERECHE;  pthread\_t tid[100];  PERECHE pair;  void\* computepairs(void\* pair) {  int n1 = atoi(((PERECHE\*)pair)->n1);  int n2 = atoi(((PERECHE\*)pair)->n2);  printf(”N1=%d N2=%d \n”, n1, n2);  }  int main(int argc, char\* argv[]) {  int i, p, n = (argc-1)/2;  for (i = 1, p = 0; p < n; i += 2, p++) {  pair.n1 = argv[i];  pair.n2 = argv[i+1];  pthread\_create(&tid[p], NULL, computepairs, (void\*)&pair);    }  for (i=0; i < n; i++)  pthread\_join(tid[i], NULL);  return 0;  } |

### Mutex /RWLock

1. Replace the mutex with a different synchronisation mechanism that has the same effect:

**R:**

**Mutex RWLOCK Binary Semaphore**

|  |  |  |
| --- | --- | --- |
| int n=0;  pthread\_mutex\_t m= PTHREAD\_MUTEX\_INITIALIZER;    void\* f(void \* a){  pthread\_mutex\_lock(&m);  n++;  pthread\_mutex\_unlock(&m);  return NULL;  }    int main(){  int i;  pthread\_t t[10];  for (i=0; i<10; i++)  pthread\_create(&t[i],0,f,0);  for (i=0; i<10; i++)  pthread\_join(t[i], NULL);    pthread\_mutex\_destroy(&m);  return 0;  } | pthread\_rwlock\_t m= PTHREAD\_RWLOCK\_INITIALIZER;  pthread\_rwlock\_wrlock(&m);  n++;  pthread\_rwlock\_unlock(&m);  pthread\_rwlock\_destroy(&m); | sem\_t m;  sem\_wait(&m);  N++  sem\_post(&m);  sem\_init(&m,0,1);  sem\_destroy(&m); |

1. Consider the code below. Discuss the use of mutex and see if there could be a more efficient way to use it.

**R:** The mutex blocks the entire code of the thread, therefore nothing is executed concurrently. The mutex must be used only for the critical section, so locked before using countE and unlocked immediately after. In this way we ensure the protection of the critical section and the concurrent execution of the other instructions.

pthread\_mutex\_lock(&m);

if (numeo[0]==’E’) countE++;

pthread\_mutex\_unlock(&m);

|  |
| --- |
| #include <pthread.h>  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #include <ctype.h>  #define MAXLINIE 1000  pthread\_t tid[100]; //we need to refer to each thread to join them  int countE=0;  pthread\_mutex\_t m= PTHREAD\_MUTEX\_INITIALIZER;  void\* ucap(void\* numei) {  pthread\_mutex\_lock(&m);  printf("Thread start: %ld ...> %s\n", pthread\_self(), (char\*)numei);  char numeo[100];  strcpy(numeo, (char\*)numei);  if ( numeo[0]>=`a` && numeo[0]<=`z`)  numeo[0]+=’A’-‘a’;  if (numeo[0]==’E’) countE++;  printf("Thread finished: %ld > %s\n", pthread\_self(), (char\*)numeo);  pthread\_mutex\_unlock(&m);  }  int main(int argc, char\* argv[]) {  int i;  for (i=1; argv[i]; i++) {  pthread\_create(&tid[i], NULL, ucap, (void\*)argv[i]);  printf("Thread created: %ld ...> %s\n", tid[i], argv[i]);  }  for (i=1; argv[i]; i++) pthread\_join(tid[i], NULL);  printf("All threads finished\n");  pthread\_mutex\_destroy(&m);  return 0;  } |

### Barriers

What will this code print?

A) When pthread\_barrier\_init(&b1, 0, 11); and we have 7 threads.

**R**: It does not print anything. Gets stuck at the barrier.

B) If we change to pthread\_barrier\_init(&b1, 0, 7); we have 7 threads.

**R:** Threads all move forward. But there is no mutex, so n will be anything between 0/1 and 7

C) If we change to pthread\_barrier\_init(&b1, 0, 3);.

**R:** Gets stuck (threads go in rounds of 3, one remains extra… ).If we had 9 threads and barrier 3, N would be anything between 0/1 and 9.

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  int n=0;  pthread\_barrier\_t b1,b2;    void\* f(void \* a){  pthread\_barrier\_wait(&b1);  n++;  return NULL;  }    int main(){  int i;  pthread\_t t[7];  pthread\_barrier\_init(&b1, 0, 11);    for (i=0; i<7; i++)  pthread\_create(&t[i], NULL, f, NULL);  for (i=0; i<7; i++)  pthread\_join(t[i], NULL);    pthread\_barrier\_destroy(&b1);  printf("%d \n", n);  return 0;  } |

1. Conditional variables

What will be printed by the following program?

A) As it is, with CONDITION=9, **R:** it always prints 10.

B) If we #define CONDITION 3, **R:** It may print anything between 4 and 10.

C) If we #define CONDITION 10? **R:** Nothing, it gets stuck waiting for the signal.

|  |
| --- |
| #include <stdio.h>  #include <pthread.h>  #include <unistd.h>  #define CONDITION 9  int n=0;  pthread\_mutex\_t m = PTHREAD\_MUTEX\_INITIALIZER;  pthread\_cond\_t c = PTHREAD\_COND\_INITIALIZER;    void\* f(void\* a){    pthread\_mutex\_lock(&m);  n++;  if (n>CONDITION)  pthread\_cond\_signal(&c);  pthread\_mutex\_unlock(&m);  return NULL;  }    int main(){  int i;  pthread\_t t[10];    for (i=0; i<10; i++)  pthread\_create(&t[i], NULL, f, NULL);    pthread\_mutex\_lock(&m);  while (n<=CONDITION)  pthread\_cond\_wait(&c, &m);  printf("%d \n", n);  pthread\_mutex\_unlock(&m);    for (i=0; i<10; i++)  pthread\_join(t[i], NULL);  return 0;  } |

### Semaphores

Given the code below:

A) What will this program print when the semaphore is binary? **R:** 1000000

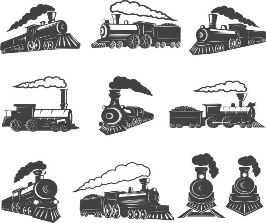
B) What will it print when sem\_init(&sem, 0, 100); ? **R:** Any number. We’ll get 999947, 1000000, 999995, etc…

C) How about when sem\_init(&sem, 0, 20); ? **R:** Same as B.

|  |
| --- |
| #include <stdlib.h>  #include <stdio.h>  #include <pthread.h>  #include <semaphore.h>  #include <unistd.h>  #define T 1000    sem\_t sem;  pthread\_t t[T];  long n=0;    void\* f(void\* v){  int i;  for (i=0; i<1000; i++){  sem\_wait(&sem);  n++;  // to test, you need to find ways to mix up thread operations. For example:  // int a=n; a++; if (i<5 && rand()<50000) sleep(rand()%2); n=a;  sem\_post(&sem);  }  return NULL;  }    int main(){  sem\_init(&sem, 0, 1);  int i=0;    for (i=0; i<T; i++)  pthread\_create(&t[i], NULL, f, NULL);  for (i=0; i<T; i++)  pthread\_join(t[i], NULL);    sem\_destroy(&sem);  printf("%lu\n",n);  return 0;  } |

## Trains (Conditional variable and Semaphores)

**Between two train stations A and B (say Cluj and Vienna) there are m trains than need to pass on n lines.** M trains enter Cluj train station, and they want to get to Vienna. Between Cluj and Vienna there are n lines, m>n, but in the train station we can have at most n trains. Trains enter Cluj at a random interval, if there is a free line they continue towards Vienna, and this takes a certain (random) amount of time. Simulate these trains passing by.

 M trains N lines à

A (Cluj) B (Vienna)

|  |  |
| --- | --- |
| **trenuriMutCond.c** | **trenuriSem.c** |
| #include <stdlib.h>  #include <pthread.h>  #include <stdio.h>  #include <unistd.h>  #include <time.h>  #define N 5  #define M 13  #define SLEEP 4  pthread\_mutex\_t mutcond;  pthread\_cond\_t cond;  int linie[N], tren[M];  pthread\_t tid[M];  int liniilibere;  time\_t start;  void\* trece(void\* tren) {  int t, l;  t = \*(int\*)tren;  sleep(1 + rand()%SLEEP); //Before=> A    pthread\_mutex\_lock(&mutcond);  printf("Moment %lu tren %d: ==> A\n", time(NULL)-start, t);  for ( ; liniilibere == 0; ) pthread\_cond\_wait(&cond, &mutcond);  for (l = 0; l < N; l++) if (linie[l] == -1) break;  linie[l] = t; // In A ocupa linia  liniilibere--;  printf("\tMoment %lu tren %d: A ==> B linia %d\n",time(NULL)-start, t, l);  pthread\_mutex\_unlock(&mutcond);  sleep(1 + rand()%SLEEP); // Trece trenul A ==> B    pthread\_mutex\_lock(&mutcond);  printf("\t \tMoment %lu tren %d: B ==>, liber linia %d\n", time(NULL)-start, t, l);  linie[l] = -1;  liniilibere++;  pthread\_cond\_signal(&cond); // In B elibereaza linia  pthread\_mutex\_unlock(&mutcond);  }  int main(int argc, char\* argv[]) {  int i;  pthread\_mutex\_init(&mutcond, NULL);  pthread\_cond\_init(&cond, NULL);  liniilibere = N;  for (i = 0; i < N; linie[i] = -1, i++);  for (i=0; i < M; tren[i] = i, i++);  start = time(NULL);  // what about &i instead &tren[i]?  for (i=0; i < M; i++) pthread\_create(&tid[i], NULL, trece, &tren[i]);  for (i=0; i < M; i++) pthread\_join(tid[i], NULL);    pthread\_mutex\_destroy(&mutcond);  pthread\_cond\_destroy(&cond);  } | #include <semaphore.h>  #include <pthread.h>  #include <stdlib.h>  #include <stdio.h>  #include <unistd.h>  #include <time.h>  #define N 5  #define M 13  #define SLEEP 4  sem\_t sem, mut;  int linie[N], tren[M];  pthread\_t tid[M];  time\_t start;  void\* trece(void\* tren) {  int t, l;  t = \*(int\*)tren;  sleep(1+rand()%SLEEP); //Before=>A    sem\_wait(&mut);  printf("Moment %lu tren %d: ==> A\n", time(NULL)-start, t);  sem\_post(&mut);  sem\_wait(&sem); // In A ocupa linia  sem\_wait(&mut);  for (l = 0; l < N; l++) if (linie[l] == -1) break;  linie[l] = t;  printf("\tMoment %lu tren %d: A ==> B linia %d\n",time(NULL)-start, t, l);  sem\_post(&mut);  sleep(1 + rand()%SLEEP); // Trece trenul A ==> B    sem\_wait(&mut);  printf("\t \tMoment %lu tren %d: B ==>, liber linia %d\n", time(NULL)-start, t, l);  linie[l] = -1;  sem\_post(&mut);  sem\_post(&sem); // In B elibereaza linia  }  int main(int argc, char\* argv[]) {  int i;  sem\_init(&sem, 0, N);  sem\_init(&mut, 0, 1);  for (i = 0; i < N; linie[i] = -1, i++);  for (i=0; i < M; tren[i] = i, i++);  start = time(NULL);  // what about &i instead &tren[i]?  for (i=0; i < M; i++) pthread\_create(&tid[i], NULL, trece, &tren[i]);  for (i=0; i < M; i++) pthread\_join(tid[i], NULL);  sem\_destroy(&sem);  sem\_destroy(&mut);  } |

# **Home Training Problems**

1. Solve problem 6 from Seminary 5 (BEST COMPUTER SCIENCE MEMES) using conditional variables: students check the winners list then, if not all the winners were listed, they wait until new winners are announced. When a sponsor announces a winner, they notify students to check the list again.

2. After implementing the conditional variable solution for the BEST COMPUTER SCIENCE MEMES contest, the announcements website server (which was a quick set-up for the contest on a server already overcrowded with loads of other student stuff… ) started to crash because all 100 participants and their 1500 colleagues, family and friends were trying to check the list at the same time when receiving a notification. Prevent this from happening by adding a semaphore that allows a maximum of 30 persons (readers) to check the list at the same time.

# **Read more**

### Deadlock

**Conditions for Deadlock**

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

**1.** Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.

**2.** Hold and wait condition. Processes currently holding resources that were granted earlier can request new resources.

**3.** No preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

**4.** Circular wait condition. There must be a circular chain of two or more processes, each of

which is waiting for a resource held by the next member of the chain.



Deadlock solutions

Read: http://nicku.org/ossi/lab/processes/posix-threads.pdf

Program 2 and 3 (with main in Program 6) present a deadlock situation.

Program 4 is the back off solution to avoid deadlock.

Program 5 is the reordering of resources solution to avoid deadlock.