البرمجة التفرعية
Parallel Programming

References

- Peter S. Pacheco, An Introduction to Parallel Programming, Morgan Kaufmann Publishers is an imprint of Elsevier 2011

البرمجة المتوازية البرمجة عن طريق Multithreading

- البرمجة متعددة الخيوط Multithreading:
  - عملية تنفيذ مؤشرات ترابط (خيوط) متعددة في وقت واحد.
  - Thread (الخط) هو في الأساس عملية فرعية خفيفة الوزن، وهي أصغر وحدة معالجة.
  - المعالجة المتعددة والخيوط المتعدد، كلاهما يستخدم لتحقيق تعدد المهام.
    - يتم استخدام مؤشرات الترابط (الخيوط) المتعددة بدلاً من المعالجة المتعددة لأن مؤشر الترابط (Thread) يشترك في منطقة ذاكرة مشتركة.
    - لا تخصص منطقة ذاكرة منفصلة، لذا فهي توفر الذاكرة، ويستغرق تبديل السياق بين الخيوط وقتًا أقل من المعالجة
  - الاستفادة من تعدد الخيوط
    - لا يمنع المستخدم لأن الخيوط مستقلة ويمكنك إجراء عمليات متعددة في نفس الوقت
    - يمكنك إجراء العديد من العمليات معًا مما يوفر الوقت
    - الخيوط مستقلة لذلك لا تؤثر على الخيوط الأخرى إذا حدث استثناء في موضوع واحد.
  - سنستخدم مكتبات Pthreads التي تعمل مع لغة C/C++ ضمن بيئة الـ VS2010
  - مكتبات POSIX thread (Pthread)
    - مكتبات POSIX thread هي واجهة تطبيقات برمجية لدعم الخيوط للغة C/C++
    - تسمح بكتابة برامج متعددة الخيوط تعمل بالتوازي .
    - مناسبة وتظهر كفاءتها في الأنظمة متعددة المعالجات (multi-processor) والأنظمة متعددة النواة (multi-core) حيث يمكن لكل خيط أن يعمل في معالج منفصل مما يزيد سرعة التنفيذ خلال المعالجة المتوازية أو الموزعة.
    - يعتبر استخدام الخيوط أقل إزعاجا من استخدام التفريع (forking) والذي يسمح بتنفيذ أكثر من عملية في وقت واحد. لأنه لن يحتاج تهيئة مساحة ذاكرة ظاهرية وبيئة لكل عملية جديدة.

PP 2024-2025

- يمكن الاستفادة من هذه المكتبات في الأنظمة ذات المعالج الواحد (uniprocessor) حيث يمكن لخيط الاستفادة من نفس المعالج والعمل فيه إذا كان الخيط المنفذ في حالة انتظار دخل أو خرج أو أي شيء آخر ليس للمعالج دخل فيه (تعدد المهام multi-task)
- يوجد العديد من النماذج الشائعة للبرامج متعددة الخيوط منها:
  - نموذج المدير/العامل (Manager/worker):
    - حيث يقوم الخيط المدير بتوزيع المهام على الخيوط الأخرى والتي تمثل العمال.
  - نموذج الأنبوب الانسيابي (pipeline) :
    - حيث يتم تقسيم المهمة إلى عمليات فرعية بحيث تعتمد كل عملية على مخرجات العملية الأخرى،
      - لكن تنفذ كل عملية في خيط منفصل (مثل تجميع السيارات)
  - الند (peer): يشبه نموذج المدير/العامل، لكنه يختلف في أن المدير بعد توزيعه المهام على العمال يشارك هو في التنفيذ
- برمجة الخيط بأمان
  - هو مقدرة التطبيقات على تنفيذ عدة خيوط بالتوازي دون تخريب البيانات المشتركة أو توليد حالة سباق (race conditions)
  - مثال: إذا كان هنالك تطبيق يحتوي على عدة خيوط وكل خيط يستدعي نفس إجرائية المكتبة (library routine)،
    - إذا افترضنا أن هذا الإجراء يقوم بتعديل بيانات عامة او موقع في الذاكرة،
      - سيحاول كل خيط تعديل هذه البيانات في نفس الوقت مما يسبب تخريبا في هذه البيانات،
      - لذلك لابد من أن يكون هنالك نوع من الوصول المؤامن لهذه البيانات حتى نحميها من التخريب وحتى يصبح الخيط آمنا.
  - لذلك يتوجب في حال استخدام إجرائيات خارجية ضمان سلامتها والتأكد من أنها آمنة وإلا يجب تجنب استخدامها.
- واجهة التطبيقات البرمجية (Pthreads API)
  - يمكن تقسيم الواجهة البرمجية إلى أربعة مجموعات رئيسية هي:
    - إدارة الخيط:
      - هي الإجرائيات التي تعمل مباشرة مع الخيوط مثل creating, detaching, joining، الخ.
    - إجرائيات الـ mutex:

PR 2024-2025

- تستخدم في التزامن وتسمى (Mutex) من الكلمتين mutual exclusion، وهي معنية بتعديل الصفات المرتبطة مع الـ mutexes.
  - المتغيرات الشرطية (condition variables):
    - هي الإجرائيات التي تهتم بالاتصالات بين الخيوط المتشاركة في mutex وهي معتمدة على شروط يضعها المبرمج.
  - التزامن (synchronization):
    - إجرائيات لإدارة حجز القراءة والكتابة.

- كل التعريفات في مكتبة الخيوط تبدأ بالكلمة pthread_ كما موضح في الأمثلة التالية:

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

- إنشاء خيط

- لإنشاء خيط نحتاج استخدام الدالة pthread_create() والتي تحتوي على المعطيات التالية:
  - المعطى الأول نحصل من خلاله على تعريف الخيط (thread identifier)
  - المعطى الثاني مؤشر إلى مكان الكائن الذي يحدد صفات الخيط،
    - إذا تم استخدام الـ null هنا فهذا يعني استخدام الصفات الافتراضية للخيط.
  - المعطى الثالث هو مؤشر إلى مكان الدالة التي ينفذها الخيط.
  - المعطى الأخير هو القيم (argument) التي نريد تمريرها إلى الدالة المنفذة داخل الخيط،

- إذا لم يكن هناك قيم يراد تمريرها إلى الدالة يمكن كتابة null في هذه الخانة.

- مثال لو كتبت شفرة الدالة بهذه الطريقة:
- pthread_create(&th_ID, NULL, th_fun, &value);
  - فهذا يعني إنشاء خيط يخزن تعريف هذا الخيط في المتغير th_ID، ويستخدم هذا الخيط الصفات الافتراضية، وينفذ هذا الخيط الدالة th_fun التي تكون مدخلاتها value
- إذا كان المطلوب إنشاء ثلاث خيوط فيجب استدعاء الدالة pthread_create() ثلاث مرات.
- يمكن استدعاء الدالة pthread_join() مع كل خيط قمنا بإنشائه لضمان انتهاء الخيوط قبل انتهاء الدالة الرئيسية main

- إنهاء خيط.

- لإنهاء خيط نستخدم الدالة pthread_cancel()
  - لكن يجب التأكد من أن الخيط الذي تريد إنهاؤه لا يستخدم موارد قبل إنهاؤه.
- مثال: إذا كان الخيط يحجز مساحة بالذاكرة وقمنا باستدعاء دالة الإنهاء pthread_cancel()
  - سنفقد مكان هذه الذاكرة وستظل محجوزة بلا فائدة (memory leak)
- الاجرائيات المستخدمة في إنشاء وإنهاء الخيوط.
  - pthread_create (thread, attr, start_routine, arg)
  - pthread_exit (status)
  - pthread_attr_init (attr)
  - pthread_attr_destroy (attr)

PP 2024-2025

- تنبيهات عن إنشاء الخيط:
  - الدالة الرئيسية main لديها خيط واحد افتراضي، بقية الخيوط على المبرمج أن يقوم بإنشائها بنفسه.
  - الأمر pthread_create ينشيء خيط ويمكن استدعائه أكثر من مرة من أي مكان داخل الكود البرمجي لإنشاء أكثر من خيط.
  - عند انشاء الخيط يمكنه بدوره إنشاء خيوط أخرى، فليس هنالك هرمية بين الخيوط.

- واصفات الخيط (Thread Attributes)
  - افتراضيا ينشأ الخيط بصفات معينة، ويمكن للمبرمج تغير بعض هذه الصفات عبر كائن الصفات (thread attribute object)
  - تستخدم pthread_attr_init و pthread_attr_destroy لتهيئة/تدمير كائن الصفات .
  - هنالك إجرائيات اخرى تستخدم لمعرفة أو تغيير صفات معينة في كائن الصفات.
  - انهاء الخيط (Terminating Threads)
    - هنالك عدة طرق لإنهاء الخيط منها:
      - رجوع الخيط من الإجرائية التي بدأ فيه (الإجرائية الرئيسية التي قامت بتهيئة الخيط)
      - استدعاء الخيط للإجرائية pthread_exit .
      - إلغاء الخيط بخيط آخر وذلك باستدعاء الإجرائية pthread_cancel .
      - إذا انتهت العملية بكاملها باستدعاء الإجرائية مثل exec أو exit .
    - يمكن استخدام الإجرائية pthread_exit للخروج من الخيط ويتم استدعاءها في نهاية الخيط عند ما نريد عمل شيء آخر.
    - إذا انتهى البرنامج الرئيسي ()main قبل الخيوط التي أنشأها وخرج بالأمر ()pthread_exit ، فإن الخيوط الأخرى ستظل تعمل.
      - أما إذا انتهت ()main فستنتهي معها كل الخيوط التي أنشاءها.
    - يستطيع المبرمج (اختياريا) تحديد حالة الانتهاء (termination status) والتي تكون مخزنة كمؤشر من النوع void في أي خيط قد يشارك في استدعاء الخيط.
    - الإجرائية ()pthread_exit لا تغلق الملفات المفتوحة داخل أي خيط وستظل مفتوحة حتى بعد انتهاء الخيط.

PP 2024-2025

- ملاحظة: في الإجرائيات التي من المتوقع انتهاء تنفيذها بصورة طبيعة يمكن الاستغناء عن pthread_exit()، ما لم يتم تمرير pass a return code back
  - لكن هنالك مشكلة وهي: عندما تكتمل main() قبل توزيع الخيوط (threads it spawned)
    - فإذا لم يتم استدعاء الـ pthread_exit() ضمنيا، عندما تكتمل main() فإن العملية (وكل خيوطها) ستنتهي.
    - باستدعاء الـ pthread_exit() في main()، فإن العملية وكل خيوطها ستبقى حية حتى ولو أكتمل تنفيذ كل الكود الموجود في main()
  - مثال: برنامج بسيط ينشئ خيط واحد

```
#include <pthread.h>
#include <stdio.h>
void * entry_point(void *arg) {
printf("Hello world!\n");
return NULL; }
int main(int argc, char **argv) {
pthread_t thr;
if(pthread_create(&thr, NULL, &entry_point, NULL)) {
printf("Could not create thread\n");
return -1; }
if(pthread_join(thr, NULL)) {
printf("Could not join thread\n");
return -1; }
return 0;
}
```

  - مثال: برنامج ينشئ 5 خيوط

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *PrintHello(void *threadid) {
long tid;
```


PP 2024-2025

```
tid = (long)threadid;
printf("Hello World! It's me, thread #%ld!\n", tid);
pthread_exit(NULL);
return NULL; }
int main (int argc, char *argv[]) {
pthread_t threads[NUM_THREADS];
int rc;
long t;
for(t=0; t<NUM_THREADS; t++){
printf("In main: creating thread %ld\n", t);
rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
if (rc){
printf("ERROR; return code from pthread_create() is %d\n", rc);
//exit(-1);
}}
pthread_exit(NULL); }
```

- تمرير قيم إلى الخيط (Passing Arguments to Threads)
  - من تابع الانشاء نلاحظ أن القيمة التي يتم تمريرها للدالة هي مخزنة في متحول، وهي قيمة واحدة.
    - ذلك لان الدالة pthread_create() تسمح بتمرير قمية واحدة (one argument) للدالة التي تنفذ الخيط.
  - لتمرير أكثر من قيمة يمكن استخدام بنية بيانات (data structure) تحتوي كل القيم المراد تمريرها (مثل السجلات أو الأصناف أو المؤشرات)، ثم وضع مؤشر هذه البنية في الدالة pthread_create()
  - مثال:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8
char *messages[NUM_THREADS];
struct thread_data {
int thread_id;
int sum;
char *message; };
```


PP 2024-2025

```
struct thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void *threadarg) {
int taskid, sum;
char *hello_msg;
struct thread_data *my_data;
my_data = (struct thread_data *) threadarg;
taskid = my_data->thread_id;
sum = my_data->sum;
hello_msg = my_data->message;
printf("Thread %d: %s Sum=%d\n", taskid, hello_msg, sum);
pthread_exit(NULL);
return NULL;
}
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum;
sum=0;
messages[0] = "English: Hello World!";
messages[1] = "French: Bonjour, le monde!";
messages[2] = "Spanish: Hola al mundo";
messages[3] = "Klingon: Nuq neH!";
messages[4] = "German: Guten Tag, Welt!";
messages[5] = "Russian: Zdravstvytye, mir!";
messages[6] = "Japan: Sekai e konnichiwa!";
messages[7] = "Latin: Orbis, te saluto!";
for(t=0;t<NUM_THREADS;t++) {
sum = sum + t;
thread_data_array[t].thread_id = t;
thread_data_array[t].sum = sum;
thread_data_array[t].message = messages[t];
printf("Creating thread %d\n", t);
rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
&thread_data_array[t]);
if (rc) {
printf("ERROR; return code from pthread_create() is %d\n", rc);
exit(-1);
}
}
system("pause");
pthread_exit(NULL);
```


PR 2024-2025

}

- الإنضمام (Joining):
  - تعتبر احدى الطرق لتنفيذ التزامن (synchronization) بين الخيوط
  - التابع pthread_join() يحجز الخيط المستدعي (calling thread)حتى ينتهي الخيط المحدد (threaded)
  - يمكن الحصول على حالة انتهاء الخيط الهدف إذا كان محددا عند استدعاء التابع pthread_exit() .
  - تتم عملية إنضمام الخيط (joining thread) مرة واحدة فقط
    - أي استدعاء الروتين pthread_join()مرة واحدة لنفس الخيط
    - يعتبر خطأ منطقي محاولة عمل أكثر من إنضمام (multiple joins) لنفس الخيط.
  - هنالك طرق أخرى للتزامن مثل mutexes و المتغيرات الفرعية (condition variables)
  - هل الخيط قابل للإنضمام أم لا (Joinable or Not؟)
    - عند إنشاء الخيط هنالك أحد صفاته توضح هل هذا الخيط قابل للإنضمام (joinable) أم لا (detached)
      - فقط الخيوط القابلة للإنضمام هي التي يمكن ضمها
      - إذا أنشيء الخيط كخيط منفصل (detached) فلا يمكن ضمه أبدا.
    - في معايير POSIX الاخيرة تم تحديد أن الخيط يجب أن يكون من النوع القابل للضم (joinable) عند إنشائه.
    - لتحديد حالة الخيط عند إنشائة هل سيكون joinable أو detached، يمكن استخدام القيمة المدخلة attr ضمن تابع الانشاء . pthread_create() :
  - خطوات تغيير خاصية الخط:
    - Declare a pthread attribute variable of the pthread_attr_t data type
    - Initialize the attribute variable with pthread_attr_init()
    - Set the attribute detached status with pthread_attr_setdetachstate()
    - When done, free library resources used by the attribute with pthread_attr_destroy()
- الإنفصال (Detaching)
  - يمكن استخدام التابع pthread_detach() لجعل الخيط منفصل حتى ولو تم انشائه قابل للانضمام (joinable)
    - لا يوجد تابع عكسي.
  - ملاحظة:

PP 2024/2025

- يجب انشاء الخط من البداية من النوع القابل للانضمام (joinable) عندما يكون الهدف البرمجي يتطلب ذلك ، الأمر الذي يسمح بالتنقلية (portability)
  - في الحالة العامة ليس كل المكتبات تنشيء الخيط من البداية قابل للانضمام (joinable by default)
- في حال كان الهدف البرمجي يتطلب أن الخيط لن يحتاج إنضمام مع خيط آخر فيمكن إنشائه في حالة المنفصل حيث يوفر ذلك تحرير بعض موارد النظام.

- يوجد العديد من التوابع والاجرائيات الأخرى المفيدة والتي يمكن استخدامها مع الخطوط:
  - التابع pthread_self : يعطي (returns) رقم خيط فريد للخيط المستدعي (calling thread)
  - التابع pthread_equal: يقارن بين رقمي خيطين (thread Ids) ويعطي 0 إذا كانا مختلفان وقيمة غير صفرية في غير ذلك.
    - يعتبر رقم تعريف الخيط كائن وبالتالي لا يمكن استخدام العلامة == لمقارنة قيم الارقام التعريفية للخيوط (thread Ids)
- متغيرات الميوتكس (Mutex Variables)
  - كلمة Mutex هي اختصار لـ "mutual exclusion"
  - تعتبر متغيرات Mutex من أهم التطبيقات التي تستخدم لتحقيق التزامن بين الخيوط (thread synchronization) وحماية البيانات المشتركة عند حدوث أكثر من أمر كتابة عليها (multiple writes)
  - يعمل المتغير mutex كإقفال (lock) لمنع الوصول إلى البيانات المشركة.
    - المعنى من وراء فكرة المتغير mutex المستخدمة في Pthreads هو أن خيط واحد فقط هو من يقفل (يمتلك) متغير mutex في أي لحظة.
      - أي إذا كان هنالك عدد من الخيوط تحاول قفل mutex واحدة فقط هي التي ستنجح. ولن يستطيع أي خيط آخر قفل (إمتلاك) هذا المتغير (mutex) ما لم يتركه الخيط الذي حصل عليه مسبقا.
  - يمكن استخدام Mutexes في الوقاية من (منع) حدوث حالات سباق ("race" conditions)
    - مثال:
      - يجب عمل mutex لغلق الرصيد (lock the "Balance") بينما يقوم الخيط باستخدام هذا البيانات المشتركة.
      - المتغيرات التي تم تحديثها تنتمي إلى مقطع حرج (critical section)

PP 2024-2025

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

- مثال:
  - مقارنة بين حالة استخدام Mutexes وعدم استخدامه

Without Mutex	With Mutex
`int counter=0;`	`/* Note scope of variable and mutex are the same */` `pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;`

`/* Function C */`
`void functionC()`
`{`
`  counter++`
`}`
	`int counter=0;`
	
	`/* Function C */`
	`void functionC()`
	`{`
	`  pthread_mutex_lock( &mutex1 );`
	`  counter++`
	`  pthread_mutex_unlock( &mutex1 );`
	`}`

Possible execution sequence

Thread 1	Thread 2	Thread 1	Thread 2
counter = 0	counter = 0	counter = 0	counter = 0
counter = 1	counter = 1	counter = 1	Thread 2 locked out. Thread 1 has exclusive use of variable `counter`
			counter = 2

- خطوات استخدام mutexكما يلي:
  - إنشاء وتهيئة المتغير . mutex
  - عدة خيوط تحاول غلق الـ mutex
  - خيط واحد فقط ينجح في امتلاك الـ mutex
  - ينفذ الخيط المالك لـ mutex التعامل مع البيانات.
  - يترك الخيط المالك الـ . mutex
  - يحصل خيط آخر على الـ mutex يؤدى عملا ما.
  - في النهاية يتم تدمير الـ . mutex
- عندما تتنافس عدة خيوط على mutex، تظل الخيوط التي لم تحصل على الـ mutex محجوزة في ذلك الوضع وتستدعى طلب الحجز بالامر "trylock" بدلا من استدعائه بالامر . "lock"

PP 2024-2025

- عند حماية بيانات مشتركة فإن مهمة المبرمج التأكد من أن كل خيط يحتاج استخدام الـ mutex قد فعل.
  - فمثلا لوكان لدينا ثلاث خيوط تريد تعديل نفس البيانات، ولكن خيط واحد فقط هو من استخدم الـ mutex، فستظل البيانات مخربة ولا فائدة من استخدام الـ mutex
- إنشاء وتدمير ال Mutexes
  - الاجرائيات المستخدمة:
    - pthread_mutex_init (mutex,attr)
    - pthread_mutex_destroy (mutex)
    - pthread_mutexattr_init (attr)
    - pthread_mutexattr_destroy (attr)
  - طريقة الاستخدام:
    - يجب الإعلان عن متغيرات الـ Mutex بالنوع pthread_mutex_t، ثم تهيئتها قبل استخدامها.
    - يوجد طريقتين لتهيئة المتغير mutex هي:
      - ساكن (Statically)، عند الاعلان عنه. مثلا:
        - pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
      - ديناميكي (Dynamically)، باستخدام التابع pthread_mutex_init()
        - يمكن إعداد صفات كائن الـ mutex وهي . attr
    - عند التهيئة يكون المتغير mutex غير محجوز (unlocked)
- غلق وفتح ال Mutexes
  - الاجرائيات المستخدمة:
    - pthread_mutex_lock (mutex)
    - pthread_mutex_trylock (mutex)
    - pthread_mutex_unlock (mutex)
  - طريقة الاستخدام:
    - التابع pthread_mutex_lock() يستخدم بواسطة الخيط للحصول على المتغير mutex(الغالق).
      - إذا كان الـ mutex مغلق بخيط آخر سيتم حجز هذا النداء حتى يتم فتح (unlocked) المتغير mutex

PP 2024-2025

- التابع pthread_mutex_trylock() يحاول غلق المتغير mutex، لكنه يعطي عبارة الخطأ busy مباشرة إذا وجد المتغير mutex مغلق.
  - This routine may be useful in preventing.
  - يكون هذا الروتين مناسب لتجنب شرط من شروط الاختناق (deadlock)
- التابع pthread_mutex_unlock() يفتح الـ mutex إذا استدعي بواسطة الخيط المالك للـ mutex، وهو مطلوب استدعاءه بعد أن ينتهي الخيط من استخدام البيانات المحمية.
- قد يحدث خطأ في الحالات التالية:
  - إذا كان الـ mutex أصلا مفتوح (unlocked)
  - إذا كان الـ mutex مملوك لخيط آخر.
  - مثال:

```
#include <pthread.h>
#include <iostream>
#include <math.h>
#define ITERATIONS 10000
// A shared mutex
pthread_mutex_t mutex;
double target;
void* opponent(void *arg) {
for(int i = 0; i < ITERATIONS; ++i) {
// Lock the mutex
pthread_mutex_lock(&mutex);
target -= target * 2 + tan(target);
// Unlock the mutex
pthread_mutex_unlock(&mutex);
}
return NULL;
}
int main(int argc, char **argv) {
pthread_t other;
target = 5.0;
// Initialize the mutex
if(pthread_mutex_init(&mutex, NULL)) {
printf("Unable to initialize a mutex\n");
return -1;
}
if(pthread_create(&other, NULL, &opponent, NULL)) {
```

PP 2024-2025

```
printf("Unable to spawn thread\n");
return -1;
}
for(int i = 0; i < ITERATIONS; ++i) {
pthread_mutex_lock(&mutex);
target += target * 2 + tan(target);
pthread_mutex_unlock(&mutex);
}
if(pthread_join(other, NULL))
{
printf("Could not join thread\n");
return -1;
}
// Clean up the mutex
pthread_mutex_destroy(&mutex);
printf("Result: %f\n", target);
system("pause");
return 0;
}
```

- المتغيرات الشرطية (Condition Variables):
  - تعتبر المتغيرات الشرطية نوع آخر يستخدم في تزامن الخيوط (synchronization)
    - بينما يعتمد التزامن في mutexes على التحكم في وصول الخيط للبيانات، يعتمد التزامن في المتغيرات الشرطية على القيمة الفعلية للبيانات.
  - بدون متغيرات شرطية يحتاج المبرمج لخيوط تختبر باستمرار (polling) هل تم تحقق الشروط أم لا ، وهذا قد يهدر الموارد ويستهلكها لأن الخيط سيكون دوما مشغولا بهذه الامور.
    - المتغير الشرطي يسمح لنا التأكد من تحقق الشرط دون الحاجة لـ polling
    - المتغيرات الشرطية تعمل مع غلق أو قفل الـ mutex .
  - طريقة استخدام المتغيرات الشرطية:
    - إنشاء وتدمير المتغيرات الشرطية
      - pthread_cond_init (condition,attr)
      - pthread_cond_destroy (condition)
      - pthread_condattr_init (attr)
      - pthread_condattr_destroy (attr)
  - طريقة استخدام المتغيرات الشرطية:
    - إنشاءوتدمير المتغيرات الشرطية
      - طريقة الاستخدام:

PP 2024-2025

- يجب الاعلان عن المتغيرات الشرطية بالنوع pthread_cond_t،
  - يجب تهيئتها قبل استخدامها. وهنالك طريقتين للتهيئة المتغيرات هما:
    - ساكنة (Statically)، عند الاعلان عنها، مثال:
    - pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;
    - ديناميكية (Dynamically) باستخدام الاجرائية pthread_cond_init()
  - قيمة تعريف المتغير الشرطي (ID) الذي ينشأ ترجع للخيط الذي يستدعيها عبر المدخل (condition parameter)
  - يمكن إعداد صفات كائن المتغير الفرعي attr .
- الاجرائية pthread_condattr_init() والاجرائية pthread_condattr_destroy() يستخدمان لإنشاء وتدمير كائنات الصفات للمتغيرات الفرعية (condition variable attribute objects)
- الاجرائية pthread_cond_destroy() تستخدم لتحرير المتغير الشرطي الذي لا نحتاج إليه.

- الانتظار والتأشير في المتغيرات الفرعية:
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)
  - طريقة الاستخدام:
- الاجرائية pthread_cond_wait() تحجز الخيط المستدعي (calling thread) حتى يتحقق الشرط المحدد (condition is signaled)
  - يجب استدعاء هذه الاجرائية عندما يكون mutex مغلق،
  - يحرر mutex تلقائيا بينما هو منتظر.
  - بعد استلام الاشارة واستيقاظ الخيط، سيغلق mutex تلقائيا ويستخدم بواسطة الخيط.
  - سيكون المبرمج مسؤولا عن فتح mutex عندما ينتهي الخيط منه.

PP 2024-2025

- الاجرائية pthread_cond_signal() تستخدم لإرسال إشارة إلى (إيقاظ) خيط آخر (الخيط الذي ينتظر المتغير الشرطي).
  - يجب أن يستدعى بعد غلق mutex، ويجب أن يفتح mutex لتكتمل الإجرائية pthread_cond_wait() عملها
- الاجرائية pthread_cond_broadcast() تستخدم بدلا من pthread_cond_signal() إذا كان هنالك أكثر من خيط محجوز في حالة الانتظار (blocking wait state)
- استدعاء الاجرائية pthread_cond_signal() قبل الروتين pthread_cond_wait() يعتبر خطأ منطقي logical error

- ملاحظة
  - غلق وفتح المتغير mutex بطريقة سليمة ضروي عند استخدام هذه الاجرائيات.
  - مثال:
- فشل غلق الـ mutex قبل استدعاء pthread_cond_wait() قد يمنعه من الحجز.
- الفشل في فتح (unlock) الـ mutex بعد استدعاء pthread_cond_signal() قد لا يسمح للاجرائية pthread_cond_wait() التي تعمل معها بالاكتمال (تبقى محجوزة)

- مثال:

```
#include <pthread.h>
#include <iostream>
#define NUM_THREADS 3
#define TCOUNT 10
#define COUNT_LIMIT 12
int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
void *inc_count(void *t) {
int i;
long my_id = (long)t;
for (i=0; i < TCOUNT; i++) {
```


PP 2024-2025

```
pthread_mutex_lock(&count_mutex);
count++;
if (count == COUNT_LIMIT) {
printf("inc_count(): thread %ld,
count = %d Threshold reached. ",my_id,
count);
pthread_cond_signal(&count_threshold_cv);
printf("Just sent signal.\n");
}
printf("inc_count(): thread %ld, count = %d, unlocking mutex\n",
my_id, count);
pthread_mutex_unlock(&count_mutex); }
pthread_exit(NULL);
return NULL; }
void *watch_count(void *t) {
long my_id = (long)t;
printf("Starting watch_count(): thread %ld\n", my_id);
pthread_mutex_lock(&count_mutex);
while (count < COUNT_LIMIT) {
printf("watch_count(): thread %ld Count= %d. Going into wait...\n", my_id,count);
pthread_cond_wait(&count_threshold_cv, &count_mutex);
printf("watch_count(): thread %ld Condition signal received. Count= %d\n", my_id,count);
printf("watch_count(): thread %ld Updating the value of count...\n", my_id,count);
count += 125;
printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
}
printf("watch_count(): thread %ld Unlocking mutex.\n", my_id);
pthread_mutex_unlock(&count_mutex);
pthread_exit(NULL);
return NULL;
}
```

PP 2024-2025

```
int main(int argc, char *argv[]) {
int i, rc;
long t1=1, t2=2, t3=3;
pthread_t threads[3];
pthread_attr_t attr;
pthread_mutex_init(&count_mutex, NULL);
pthread_cond_init (&count_threshold_cv, NULL);
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
pthread_create(&threads[0], &attr,
watch_count, (void *)t1);
pthread_create(&threads[1], &attr,
inc_count, (void *)t2);
pthread_create(&threads[2], &attr,
inc_count, (void *)t3);
for (i = 0; i < NUM_THREADS; i++) {
pthread_join(threads[i], NULL);
}
printf ("Main(): Waited and joined with %d threads. Final value of count = %d.
Done.\n",
NUM_THREADS, count);
pthread_attr_destroy(&attr);
pthread_mutex_destroy(&count_mutex);
pthread_cond_destroy(&count_threshold_cv);
system("pause");
pthread_exit (NULL);
}
```

PP 2024-2025

PP 2024-2025