Tutorial 4 - Semaphore

COMP3230 Principle of Operating System

Huang Songlin (huangs0@hku.hk), 2023-10-26

Agenda

1. Recap Tutorial 3
2. Semaphore vs Conditional Variable
3. Multi-Threading Debugging

Recap Tutorial 3

- POSIX Thread (pthread.h)
    - thr_func, create, join, exit
- Atomic & Mutex Lock
- Conditional Variable to be covered later

POSIX Thread

Reminder: In code: `#include <pthread.h>` / In compile: `gcc -pthread` (or functions not found)

Thread Function: **void *thr_func(void *arg)**

Question: Difference Between **void** and **void***?

Thread Creation: **pthread_create(**pthread_t* id, NULL, **void *thr_func(void *arg)**, void* *arg*)

Question: What's pthread_t?

Thread Exit: pthread_exit(void* exit_val)

Question: What if int a = 1 and pthread_exit((void*) a);?

Thread Termination: **pthread_join(**pthread_t id, (void**) exit_val): **BLOCKED CALL**

Practical Question:

- Which scope can thread access?

Answer:

- Which scope independent for each thread?

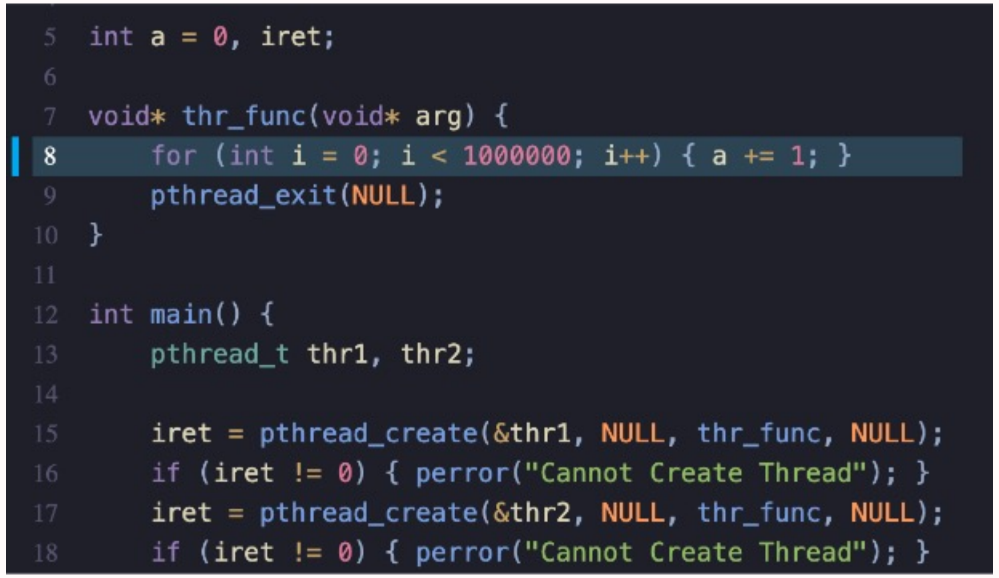
Answer:

- How thread collects parameter and return?

Answer:

Atomic & Mutex Lock

Race Condition: **multiple threads simultaneously read/write the shared resources (memory)**



```c
int a = 0, iret;

void* thr_func(void* arg) {
    for (int i = 0; i < 1000000; i++) { a += 1; }
    pthread_exit(NULL);
}

int main() {
    pthread_t thr1, thr2;

    iret = pthread_create(&thr1, NULL, thr_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread"); }
    iret = pthread_create(&thr2, NULL, thr_func, NULL);
    if (iret != 0) { perror("Cannot Create Thread"); }
```

Correct: 2000000, Actual: 1016738

Soln: one and only one can use shared resource

Basic: **Atomics** like ***__atomic_add_fetch***

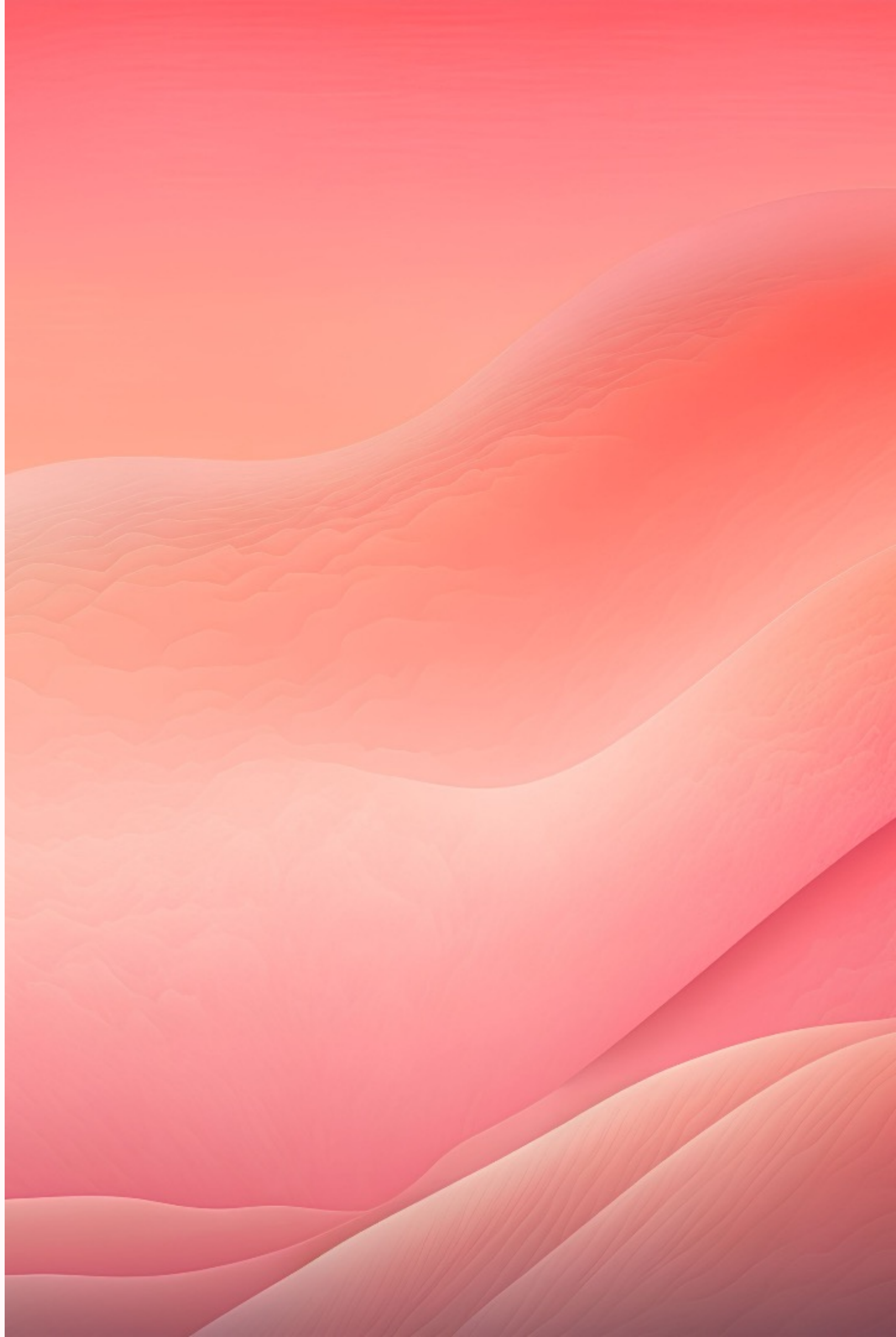
Hardware support (Lock Memory / Cache)

Mutex Locks - More customisable soln

1. Creation:
    1. Create variable by pthread_mutex_t lock;
    2. Init by pthread_mutex_init(&lock, NULL);
2. Lock & Unlock
    1. **BLOCKED** Lock by pthread_mutex_lock(&lock);
    2. **UNBLOCKED** Lock (return 0 if lock) by pthread_mutex_trylock(&lock);
    3. Unlock by pthread_mutex_unlock(&lock);
3. Destroy by pthread_mutex_destroy(&lock);

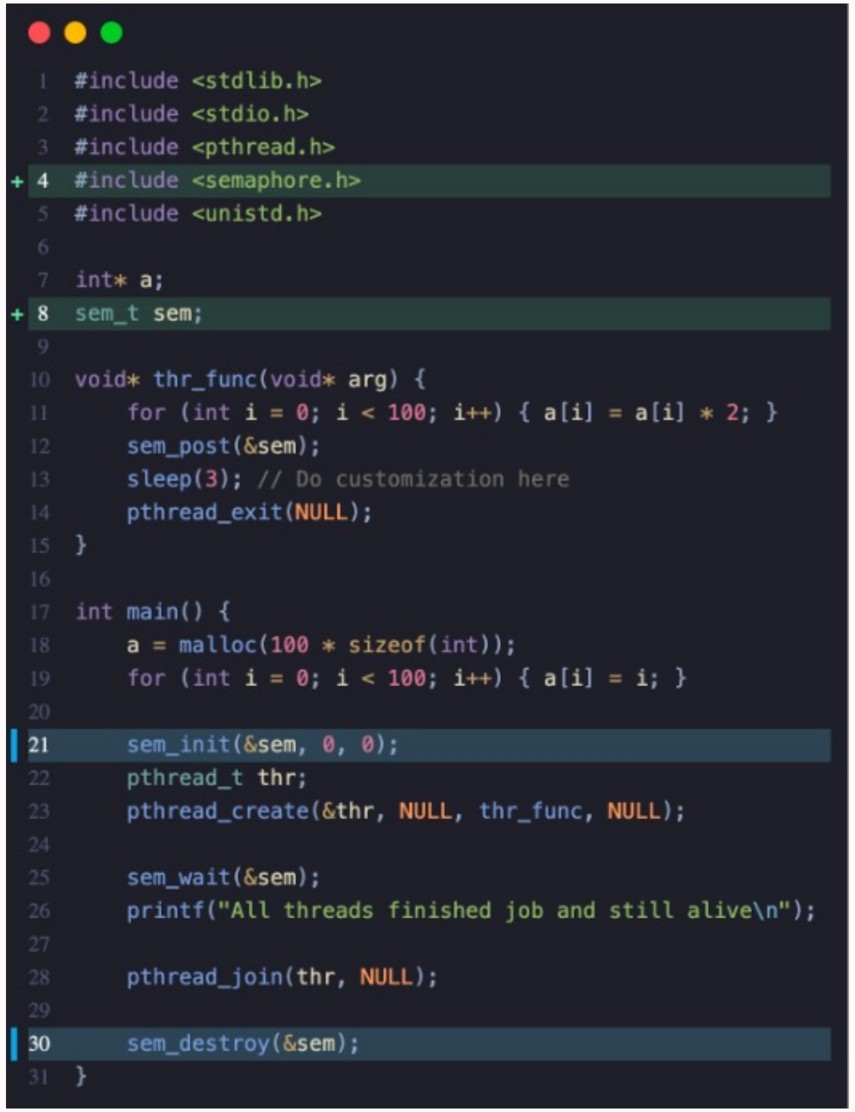
Semaphore

- <semaphore.h> API
  - Unnamed / Named - Init / Destroy
  - Wait / Post
- Case Study: Semaphore vs Cond Variable



Semaphore API - Unnamed

Reminder: in code `#include <semaphore.h>` / in compile `gcc -pthread`



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

int* a;
sem_t sem;

void* thr_func(void* arg) {
    for (int i = 0; i < 100; i++) { a[i] = a[i] * 2; }
    sem_post(&sem);
    sleep(3); // Do customization here
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    sem_init(&sem, 0, 0);
    pthread_t thr;
    pthread_create(&thr, NULL, thr_func, NULL);

    sem_wait(&sem);
    printf("All threads finished job and still alive\n");

    pthread_join(thr, NULL);

    sem_destroy(&sem);
}
```

Create Semaphore

First create variable sem_t sem; Then init by:

int sem_init(sem_t* *sem*, int *pshared*, int *value*)

- sem_t* *sem*: pointer of semaphore to be init
- int *pshared*: indicates if the semaphore is:
  - Local to this process (pshared=0)
  - shared among processes (pshared=1)
- unsigned int *value*: the initial value of sem

Destroy Semaphore

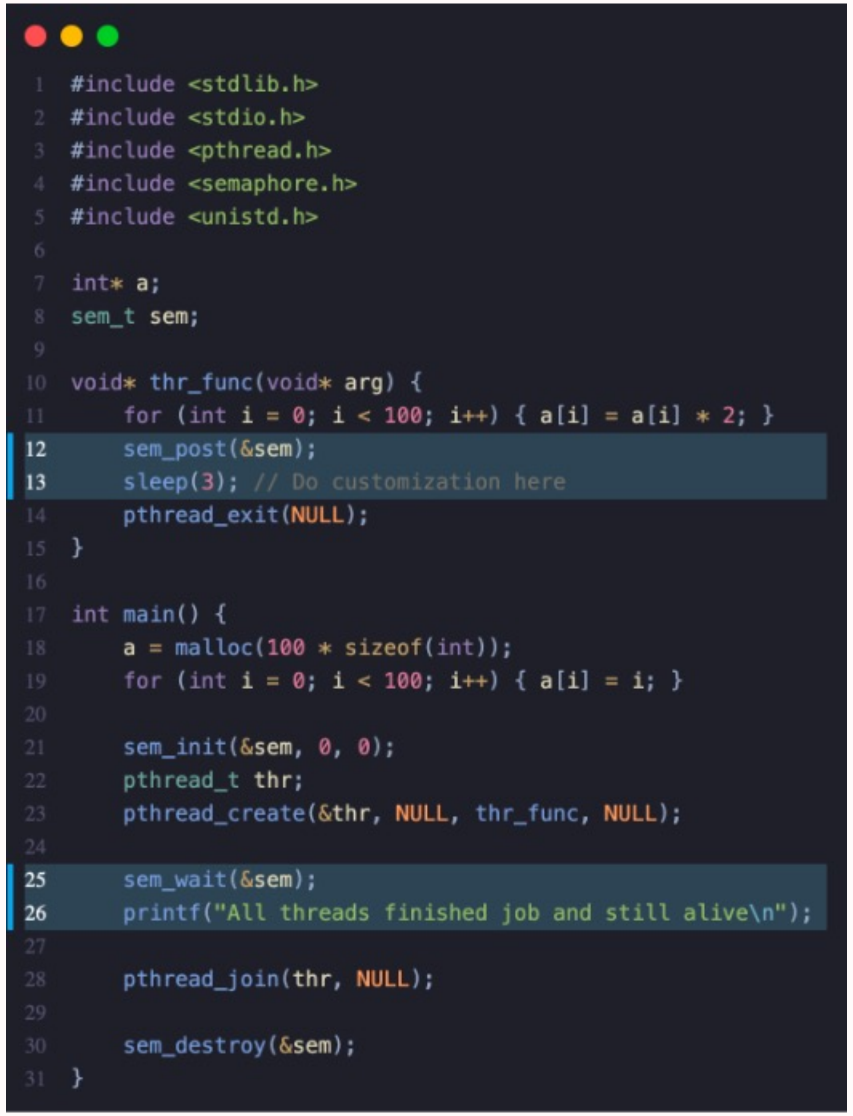
int sem_destroy(sem_t* *sem*)

- sem_t* *sem*: pointer of sem to be destroyed

P.S. Return 0 if succeed and non-zero if failed

Semaphore API - Post / Wait

Two **Atomic Operation**: Wait: P(&sem), sem_wait(&sem) / Post: V(&sem), sem_post(&sem)



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

int* a;
sem_t sem;

void* thr_func(void* arg) {
    for (int i = 0; i < 100; i++) { a[i] = a[i] * 2; }
    sem_post(&sem);
    sleep(3); // Do customization here
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    sem_init(&sem, 0, 0);
    pthread_t thr;
    pthread_create(&thr, NULL, thr_func, NULL);

    sem_wait(&sem);
    printf("All threads finished job and still alive\n");

    pthread_join(thr, NULL);

    sem_destroy(&sem);
}
```

Wait: P(&sem)

int sem_wait(sem_t* *sem*), try **sem -= 1**, and:

- if sem > 0, perform sem -= 1 and return
- if sem = 0, blocked until sem -= 1 can be performed (sem > 0) then do sem -= 1 return

Post: V(&sem)

int sem_post(sem_t* *sem*), do **sem += 1**, and:

- if no thread is waiting, simply return
- if some thread(s) is waiting, wake up one(to do **sem-=1** and return,so exactly one thread)

Comparison with Cond Variable

Seems pthread_cond_wait ≈ sem_wait, and pthread_cond_signal ≈ sem_post? Hint: sem = 0 is "signal"

int pthread_cond_wait(pthread_cond_t**cond*, ...)

Being blocked until receiving signal of *cond*

int sem_wait(sem_t* *sem*)

if **sem <= 0 Being blocked** ,

if **sem>0** do **sem -= 1** and return

int pthread_cond**_signal(pthread_cond_t**cond***)**

Wake up one thread in waiting queue and

do nothing if queue empty

int sem_post(sem_t* *sem*)

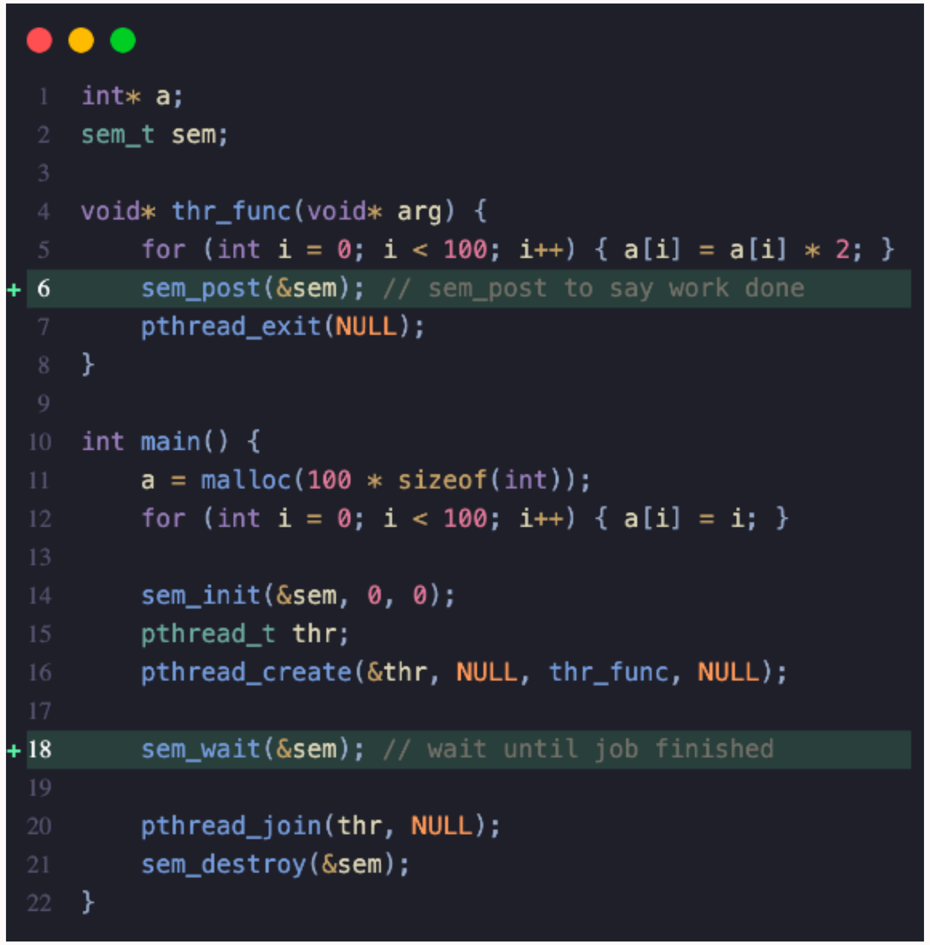
Do **sem += 1** and **Wake up one** waiting thread and **do nothing if queue empty (sem>0** before**)**

Generally, **Semaphore is Conditional Variable + an integer state**.

Case Study - no (int) state means what?

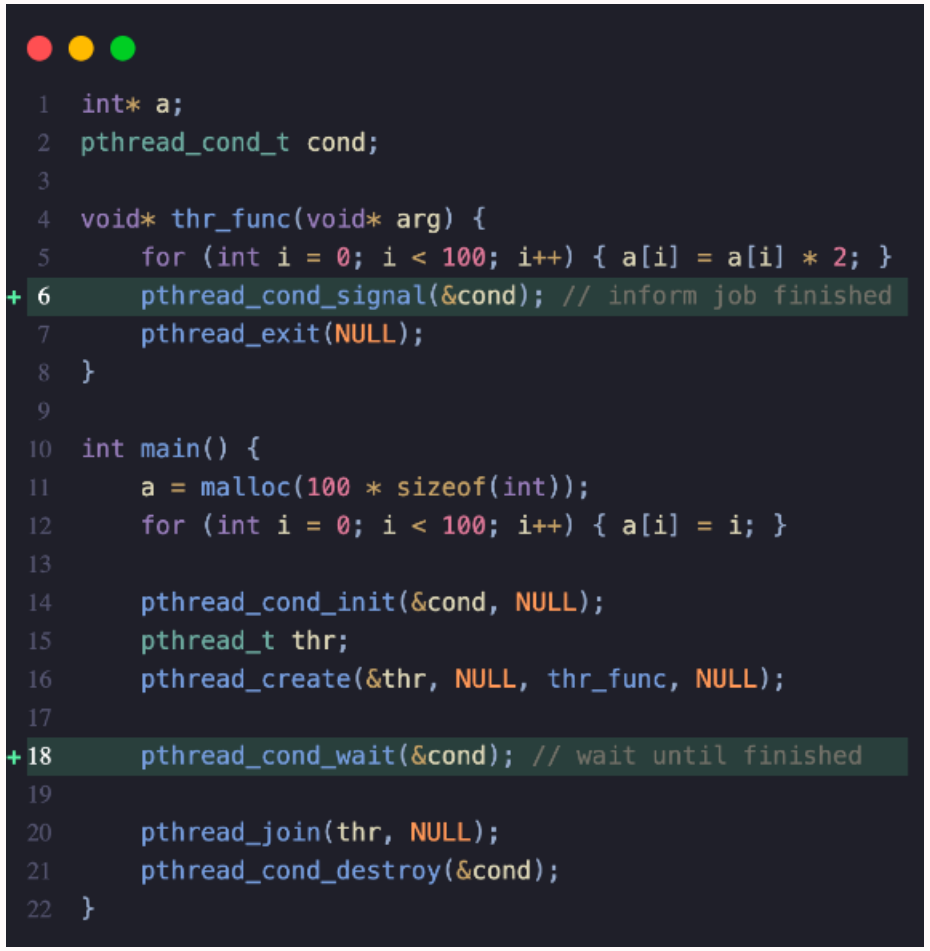
Let the only (child) thread do sth and inform parent work is done but not pthread_join?

Semaphore Implementation



```c
int* a;
sem_t sem;

void* thr_func(void* arg) {
    for (int i = 0; i < 100; i++) { a[i] = a[i] * 2; }
    sem_post(&sem); // sem_post to say work done
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    sem_init(&sem, 0, 0);
    pthread_t thr;
    pthread_create(&thr, NULL, thr_func, NULL);

    sem_wait(&sem); // wait until job finished

    pthread_join(thr, NULL);
    sem_destroy(&sem);
}
```

Replacing sem_post and sem_wait:



```c
int* a;
pthread_cond_t cond;

void* thr_func(void* arg) {
    for (int i = 0; i < 100; i++) { a[i] = a[i] * 2; }
    pthread_cond_signal(&cond); // inform job finished
    pthread_exit(NULL);
}

int main() {
    a = malloc(100 * sizeof(int));
    for (int i = 0; i < 100; i++) { a[i] = i; }

    pthread_cond_init(&cond, NULL);
    pthread_t thr;
    pthread_create(&thr, NULL, thr_func, NULL);

    pthread_cond_wait(&cond); // wait until finished

    pthread_join(thr, NULL);
    pthread_cond_destroy(&cond);
}
```

Case Study (Contd) Different Order?

Main thread and child thread executed simultaneously → order between post and wait is arbitrary.

If wait before post?

1. Main Thread call sem_wait(&sem), try sem -=1 → find sem = 0, being blocked
2. Child Thread call sem_post(&sem), do sem += 1→ sem = 1 → wake up main
3. Main Thread being waked up by child, finish sem -= 1 → sem = 0 → all done 🥳

1. Main Thread call cond_wait(&cond), put itself in waiting queue and being blocked
2. Child Thread call sem_post(&cond), wake up one from queue → wake up main
3. Main Thread being waked up and return → all done 🥳

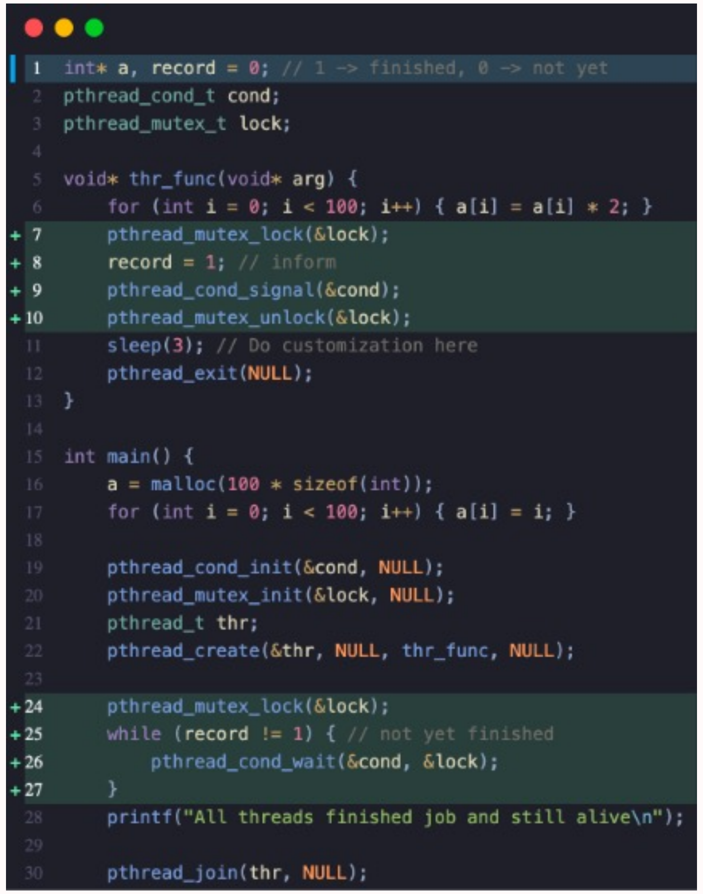
If post before wait?

1. Child Thread call sem_post(&sem), do sem += 1 → sem = 1 → queue is empty, do nothing and return
2. Main Thread call sem_wait(&sem), try sem -= 1 → now sem = 1 > 0 →do sem -= 1 and return → all done 🥳

1. Child Thread call cond_signal(&cond), try to wake up one from waiting queue → queue is empty, do nothing and return
2. Main Thread call cond_wait(&cond), put itself in waiting queue and being blocked → **No body will wake it up** 🥺

Problem of Standalone Conditional Variable: Signal is stateless and has no record…

Case Study (Soln) Add some record?



```
  1  int* a, record = 0; // 1 -> finished, 0 -> not yet
  2  pthread_cond_t cond;
  3  pthread_mutex_t lock;
  4
  5  void* thr_func(void* arg) {
  6      for (int i = 0; i < 100; i++) { a[i] = a[i] * 2; }
+ 7      pthread_mutex_lock(&lock);
+ 8      record = 1; // inform
+ 9      pthread_cond_signal(&cond);
+10      pthread_mutex_unlock(&lock);
 11      sleep(3); // Do customization here
 12      pthread_exit(NULL);
 13  }
 14
 15  int main() {
 16      a = malloc(100 * sizeof(int));
 17      for (int i = 0; i < 100; i++) { a[i] = i; }
 18
 19      pthread_cond_init(&cond, NULL);
 20      pthread_mutex_init(&lock, NULL);
 21      pthread_t thr;
 22      pthread_create(&thr, NULL, thr_func, NULL);
 23
+24      pthread_mutex_lock(&lock);
+25      while (record != 1) { // not yet finished
+26          pthread_cond_wait(&cond, &lock);
+27      }
 28      printf("All threads finished job and still alive\n");
 29
 30      pthread_join(thr, NULL);
```

If Wait Before Post?

1. record = 0 != 1 so Main Thread call cond_wait(&cond), put itself in waiting queue, being blocked and release lock
2. Child Thread acquire lock, update record = 1 and call cond_signal(&cond) → wake up main and release lock
3. Main Thread being waked up and found record = 1 so leave loop → all done 🥳

Question: Will single mutex_lock helps?

Solution

Use a variable to record cond_signal operation. (e.g., **an integer**)

This variable is shared among threads
→ Need **Mutex Lock** to protect thread safety:

- Line 7-10, when cond_signal, i.e., update, also change the state. Both signal and change shall be protected by Mutex Lock.
- Line 24-27, when cond_wait, i.e., receiving update, add **while (record ≠ 1)** to check record. Shall be protected by Mutex Lock → **pthread_cond_wait has Mutex Lock as the 2nd parameter**

If Post Before Wait?

1. Child Thread acquire lock, update record = 1, call cond_signal(&cond) → found queue empty and release lock
2. Main Thread acquire lock and found record = 1 so leave loop → all done 🥳

Problem Solved!

But Cond Variable + (Integer) State = ______?

Other Semaphore APIs

Get the Semaphore Value

int sem_getvalue(sem_t* *sem*, int* *sval*)

Place the current value of *sem* to *sval*.

int **sem_trywait**(sem_t* *sem*)

The unblocked version of sem_wait, will:

- return 0 if sem -= 1 succeed, i.e., sem > 0
- return non-zero if failed to do so

Named Semaphore (For Your Info)

For safety, macOs only support named semaphore.

sem_t* sem_open(**const char*** ***name***, **int** ***oflag***, **mode_t** ***mode***, **int** ***value***)

Open named semaphore by specifying name,mode, named semaphore is used for multi-processing

- int sem_close(sem_t* *sem*)
- int sem_unlink(sem_t * *sem*)

Note: Not recommending using semaphore in macOs → Underlying implementation is different

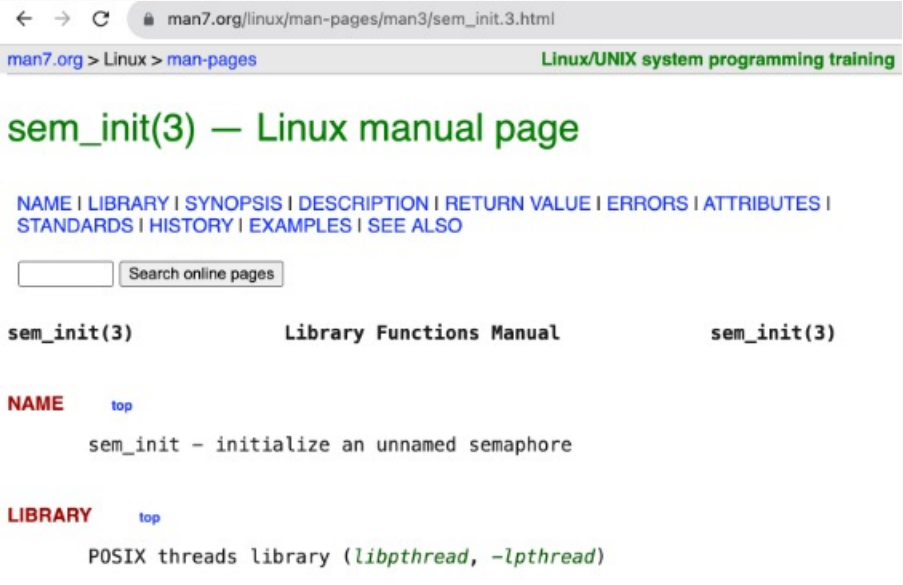
Wanna know more?

In Linux, all these underlying API is provided by Linux Kernel and GNU C Library (glibc)

Search API Documentations

Recommended way → Linux Manual Page

Search **man [API]**, Go to link with **man7.org**


man7.org/linux/man-pages/man3/sem_init.3.html
man7.org > Linux > man-pages
Linux/UNIX system programming training
Search online pages

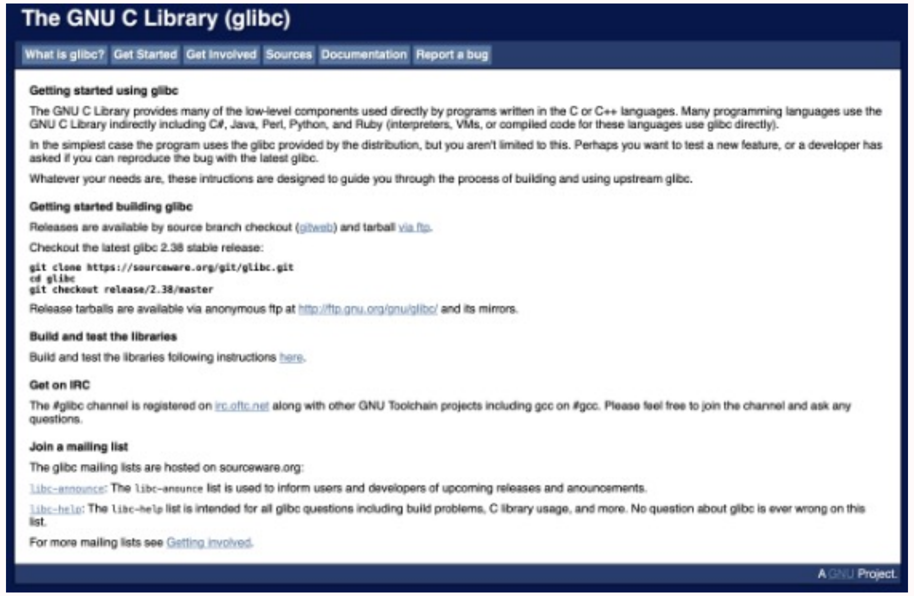
sem_init(3) — Linux manual page

NAME | LIBRARY | SYNOPSIS | DESCRIPTION | RETURN VALUE | ERRORS | ATTRIBUTES | STANDARDS | HISTORY | EXAMPLES | SEE ALSO

```
sem_init(3)              Library Functions Manual              sem_init(3)

NAME        top

       sem_init - initialize an unnamed semaphore

LIBRARY        top

       POSIX threads library (libpthread, -lpthread)
```

Search the source code

For corner cases, if don't violates the Principle of Operating System, no API record. Depends on actual implementation in Linux / glibc:



Getting started using glibc

The GNU C Library provides many of the low-level components used directly by programs written in the C or C++ languages. Many programming languages use the GNU C Library indirectly including C#, Java, Perl, Python, and Ruby (interpreters, VMs, or compiled code for these languages use glibc directly).

In the simplest case the program uses the glibc provided by the distribution, but you aren't limited to this. Perhaps you want to test a new feature, or a developer has asked if you can reproduce the bug with the latest glibc.

Whatever your needs are, these intructions are designed to guide you through the process of building and using upstream glibc.

Getting started building glibc

Releases are available by source branch checkout (gitweb) and tarball via ftp.

Checkout the latest glibc 2.38 stable release:

```
git clone https://sourceware.org/git/glibc.git
cd glibc
git checkout release/2.38/master
```

Release tarballs are available via anonymous ftp at http://ftp.gnu.org/gnu/glibc/ and its mirrors.

Build and test the libraries

Build and test the libraries following instructions here.

Get on IRC

The #glibc channel is registered on irc.oftc.net along with other GNU Toolchain projects including gcc on #gcc. Please feel free to join the channel and ask any questions.

Join a mailing list

The glibc mailing lists are hosted on sourceware.org:

libc-announce: The libc-anounce list is used to inform users and developers of upcoming releases and anouncements.

libc-help: The libc-help list is intended for all glibc questions including build problems, C library usage, and more. No question about glibc is ever wrong on this list.

For more mailing lists see Getting involved.

Can also check C Language APIs

Take Away

- Semaphore is widely used for Synchronization
    - Creation & Destroy: `sem_t, sem_init, sem_destroy`
    - Post & Wait: `sem_post, sem_wait`
    - Semaphore has an **atomic-protected integer state** as an record
- Conditional Variable is stateless → signal is just one-pass
    - Standalone Cond Variable **succeed Wait→Signal** but **fail Signal→Wait**
    - **Adding shared variables** helps → **Always used with mutex_lock**
- Think Questions — Previously we discussed one-to-one sync, and how about:
    - One-to-many Sync?
    - Many-to-One Sync? → (while instead of if)
    - Many-to-Many Syn c? → (while instead of if)

Debugging Multi-Threading Program

- VS Code Setup
- Sample Debugging