به نام خدا

آزمایشگاه سیستم های عامل

سوال اول

استاد: دکتر شهاب الدین نبوی

اعضای گروه:

حسنا بشیریان

سید محمدرضا حسینی

امیر مسعود شاکر

دانیال علی عظیمی

سید عباس میرقاسمی

مقدمه

در این گزارش به روش پیشنهادی با توجه به رویکرد Detection and Recovery برای غلبه بر شرایط بن بست در ساختار سیستم عامل pintos میپردازیم.

ابتدا بن بست (Deadlock) را تعریف میکنیم.
میتوان گفت Deadlock به وضعیتی گفته میشود که در آن چند process نیازمند منابعی هستند که توسط همدیگر block شده اند.

چهار عامل وجود دارند که میتوانند باعث به وجود آمدن Deadlock شوند. این عوامل عبارتند از:

1. Mutual Exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

برای برطرف کردن Deadlock، چهار رویکرد وجود دارد:

1. Prevention
2. Detection and Recovery
3. Deadlock Avoidance
4. Ignore

در رویکرد prevention، سعی میکنیم یکی از چهار عامل ایجاد Deadlock را از بین ببریم.

این کار میتواند باعث به وجود آمدن مشکلاتی شود.

به عنوان مثال اگر سعی کنیم عامل No preemption را حذف کنیم، ممکن است یک process در حال نوشتن در یک فایل باشد و preempt کردن منابع در دسترس آن میتواند مشکل ایجاد کند.

رویکرد Detection and Recovery رویکردی است که قرار است در این گزارش به طور مفصل به آن بپردازیم و روش پیشنهادی خود را بر مبنای آن توضیح دهیم.

به طور خلاصه ما ابتدا Deadlock را تشخیص میدهیم و در صورت وقوع آن، recovery انجام میدهیم.

رویکرد بعدی Deadlock Avoidance است که در آن سعی میکنیم Scheduling را طوری انجام دهیم که هیچ موقع دچار Deadlock نشویم. مانند الگوریتم Banker.

انجام این کار هزینه زیادی میبرد و نیازمند آن هستیم که از قبل بدانیم کدام process به چه منبعی نیاز دارد.

روش آخر Ignore است که در واقع راه حلی برای برطرف کردن Deadlock پیشنهاد نمیدهد و در صورت وقوع Deadlock، نیازمند عمل reset هستیم.

این روش در سیستم عامل هایی که احتمال وقوع Deadlock در آنها کم است مورد استفاده قرار میگیرد.

الگوریتم های تشخیص Deadlock:

برای اینکه یک سیستم detection قابل استفاده باشد باید الگوریتم مورد استفاده در آن بهینه باشد ولی در عین حال به اندازه ای کامل باشد که Deadlock را به درستی تشخیص دهد.

بر اساس مواردی که در کتاب silberschatz گفته شده است، میتوان دو سناریو کلی برای Deadlock در نظر گرفت :

- Single instance resources
- Multiple instance resources

در حل مسئله، ما حالت single instance را در نظر گرفته ایم. این روش بر اساس گراف میباشد و مبتنی بر پیدا کردن حداقل یک حلقه در داخل گراف اختصاص منابع میباشد.

الگوریتم های تشخیص حلقه زیادی وجود دارند ولی همه آنها بر اساس پیمایش گراف هستند و عموما از الگوریتم DFS استفاده میکنند و روی این الگوریتم پیاده میشوند.

برای انتخاب یک الگوریتم مناسب سه پارامتر زمان محاسبه، حافظه مورد نیاز و پیچیدگی در فهم و پیاده سازی را در نظر گرفتیم و در عین حال در حالاتی که تفاوت زیادی وجود نداشت سعی کردیم کامل ترین الگوریتم را انتخاب کنیم.

منظور ما از کاملترین الگوریتم در این مسئله، الگوریتمی است که بیشترین اطلاعات مفید و قابل استفاده را به ما میدهد که در عین حال زمان اجرای منطقی داشته باشد.

دلیل این دیدگاه در بخش آخر که مربوط به الگوریتم recovery و انتخاب victim است، دیده میشود و بر اساس این است که یک گراف میتواند بیش از یک حلقه داشته باشد و اگر امکانش وجود دارد این حلقه ها را شناسایی کرد و سعی کرد آنها از بین برد تا سیستم از حالت Deadlock خارج شود.

قبل از اینکه به الگوریتم انتخاب شده بپردازیم به کاندیداهای دیگر که به دلایل مختلف رد شدند میپردازیم.

الگوریتم های کاندید برای تشخیص Deadlock:

الگوریتم تشخیص حلقه با استفاده از مجموعه مجزا :

در این الگوریتم با استفاده از ساختمان داده disjoint set با استفاده از عملیات های union و find سعی می شود که وجود حلقه تشخیص داده شود.

این الگوریتم، هر یک از بخش های متصل در گراف را به صورت یک مجموعه در نظر میگیرد و بر اساس این دیدگاه شروع به قرار دادن parent برای هر یک از node ها میکند و اگر دو node در گراف دارای یک parent باشند تشخیص میدهیم که حلقه وجود دارد.

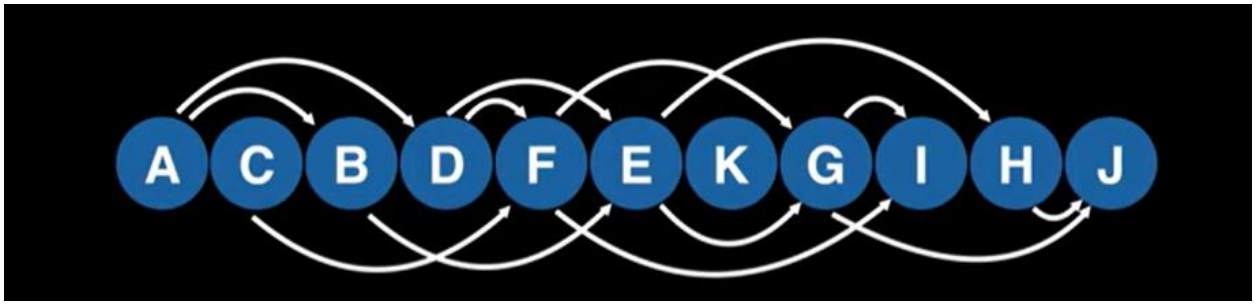
این الگوریتم دارای زمان محاسباتی خطی میباشد و با بهبود پیاده سازی ساختمان داده میتوان آن را به زمان لگاریتمی هم کاهش داد.

دلیل انتخاب نشدن این الگوریتم غیر قابل استفاده بودن آن در مسئله می باشد چون فقط در گراف های بدون جهت قابل استفاده است.

الگوریتم تشخیص حلقه با استفاده topological sort :

این الگوریتم مبتنی بر DFS است و نیازمند آن است که گراف را به طور کامل ببیند تا بتواند تشخیص حلقه دهد.

الگوریتم اصلی برای تشخیص حلقه با استفاده از مرتب سازی انجام میشود. به صورتی که اگر در یک خط مستقیم تمامی گره ها قرار بگیرند، تمامی یال ها باید به یک سمت حرکت کنند. مانند شکل زیر :



در واقع بخش تشخیص حلقه در خود DFS انجام میگیرد (مثلا با استفاده از روش رنگ کردن گره ها).

ولی پس از بدست آوردن ترتیب میتوان از این لیست مرتبط شده در تشخیص node هایی که در حلقه تاثیر گذارهستند استفاده کرد. (مانند node های میانی) و آنها را برای victim انتخاب کرد.

یک روش پیشرفته تر برای این الگوریتم موجود است که میتواند نیاز به جستجو روی کل گراف را تا حد خوبی از بین ببرد اما بخاطر سختی در پیاده سازی و پیچیدگی مفهومی آن فقط یک اشاره به آن میشود.

نام این الگوریتم Dynamic cycle detection for lock ordering میباشد و در کتابخانه هایی مانند tensorflow و abseil برای تشخیص حلقه در گراف مورد استفاده قرار گرفته است و با نگهداری یک topological order از وضعیت فعلی نخ ها سعی میکند فضای search را فقط به یک زیر فضا از فضای کل گراف محدود کند و سعی میکند که order محاسباتی را با این کار بهبود دهد.

از نظر زمان محاسباتی این الگوریتم مشابه DFS است و این زمان میتواند در اجرا های متوالی کاهش یابد :

Classic topological sort : O(|V| + |E|)

Worst case dynamic topological sort : O(|V| + |E|)

به دلیل احتمالاتی بودن روش پویا رسیدن به یک order دقیق پیچیده میباشد.

از نظر حافظه اضافی :

Classic Topological sort : O(N)

Dynamic Topological sort : O(N)

الگوریتم kosaraju :

این الگوریتم برای تشخیص Strongly connected component های گراف استفاده می شود.

این الگوریتم شامل دو عملیات DFS است.

روش تشخیص حلقه یا دور در این الگوریتم با استفاده از تشخیص SCC هایی با طول بیشتر از 1 میباشد.

نحوه تشخیص حلقه در بخش بعدی بیشتر توضیح داده میشود اما به طور مختصر میتوان گفت که با فرض شروع از گره u به گره v اگر حلقه موجود باشد باید بتوان از v به u رفت و این منطق است که باعث میشود DFS را بر روی گراف بیش از یکبار اجرا کنیم و بهینگی آن را نسبت به الگوریتم های دیگر کاهش میدهد.

پیچیدگی زمانی :

O(|V| + |E|)

حافظه مصرفی :

O(N)

الگوریتم Deadlock Detection

دو نوع الگوریتم Deadlock Detection وجود دارد.

یکی single instance detection که مبتنی بر گراف است و دیگری multiple instance detection.

الگوریتم پیشنهادی ما از نوع اول است.

برای تشخیص Deadlock ها، باید حلقه هایی که در گراف وجود دارند را شناسایی کنیم.

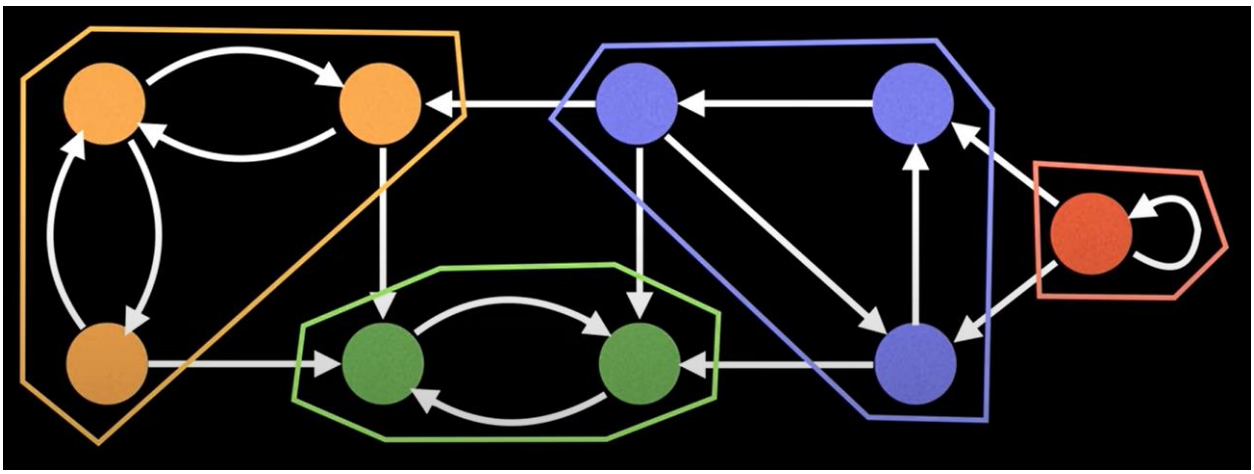
الگوریتم ما که مبتنی بر DFS است، Tarjan نام دارد.

این الگوریتم Strongly Connected Components (SCC) ها را در یک گاف جهت دار پیدا میکند.

هر SCC، مجموعه ای از راس ها است که در آن از هر راس، به راس های دیگر مسیر وجود دارد و بین دو راس دلخواه u,v، مسیر از u به v و همچنین از v به u وجود دارد.

از تعدادی راس است که میتوان با پیمایش در آن از هر راس، به خود آن رسید.

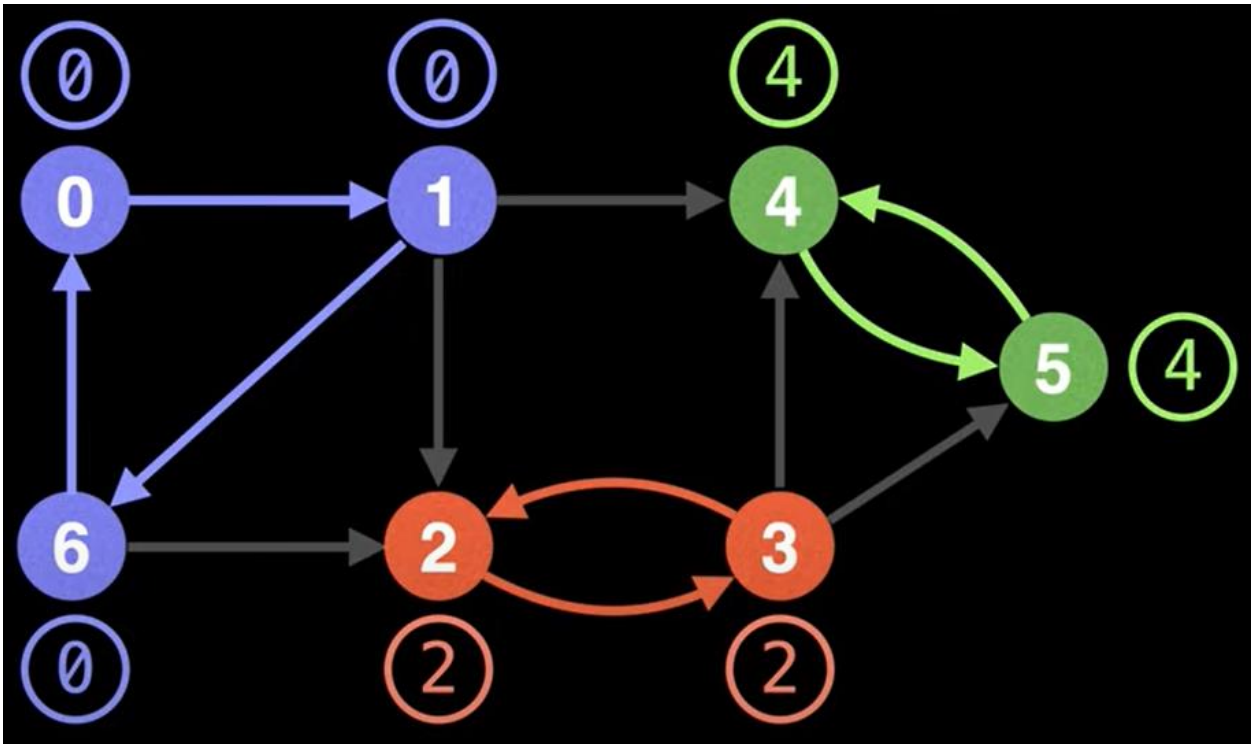
در گراف زیر چهار SCC مشخص شده است:



حال مفهموم Low-Link Value را تعریف میکنیم.

Low-Link Value برای هر راس، برابر کمترین id است که میتوانیم با انجام پیمایش DFS با شروع از آن راس به آن برسیم.

شکل زیر Low-Link Value را برای هر راس گراف نشان میدهد (عدد بالای هر راس در دایره):



مشاهده میکنیم که راس هایی که دارای یک Low-Link Value مشترک هستند، با همدیگر یک SCC تشکیل میدهند.

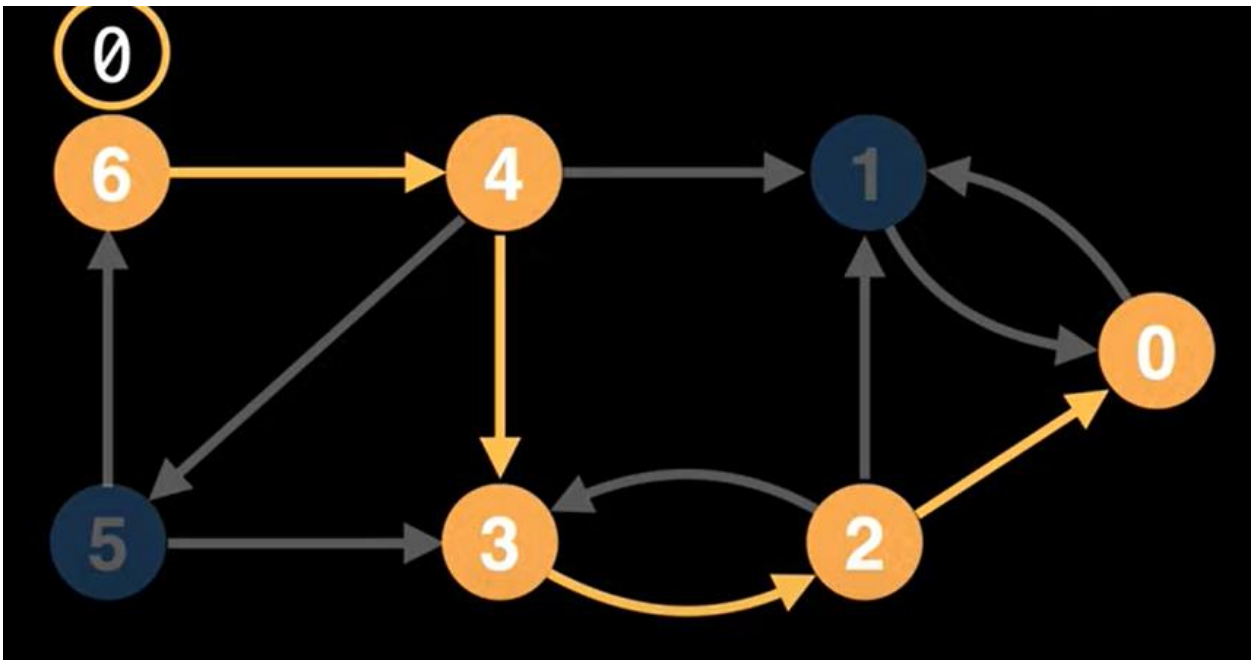
بنابراین ممکن است به این نتیجه برسیم که برای پیدا کردن SCC ها در یک گراف، میتوانیم Low-Link Value را برای هر راس حساب کنیم و راس های دارای Low-Link Value مشترک را در یک SCC قرار دهیم.

اما این کار به این سادگی قابل انجام نیست و همیشه درست جواب نمیدهد.

چرا که راس شروع ما در پیمایشی که انجام میدهیم، در نتیجه نهایی اثرگذار است.

در عکس بالا ابتدا از راس 0 که در بالا سمت چپ قرار گرفته شروع به پیمایش کردیم.

اگر در همین گراف، راس 0 را سمت راست ترین گره در نظر بگیریم و از آن شروع به پیمایش کنیم، به نتیجه زیر میرسیم:

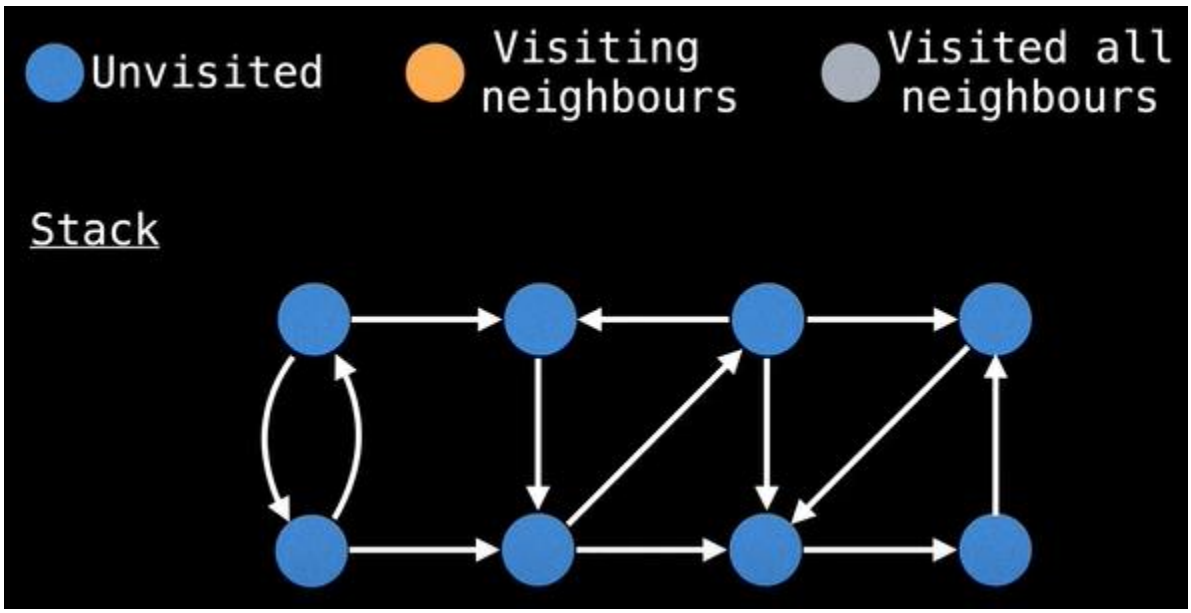


همانطور که مشاهده میشود، با شروع از راس صفر، Low-Link Value برای همه راس های نارنجی برابر صفر شده است. اما میدانیم که این راس ها یک SCC را تشکیل نمیدهند.

همچنین اگر ادامه دهیم، Low Link Value برای همه راس ها صفر میشود که نادرست است.

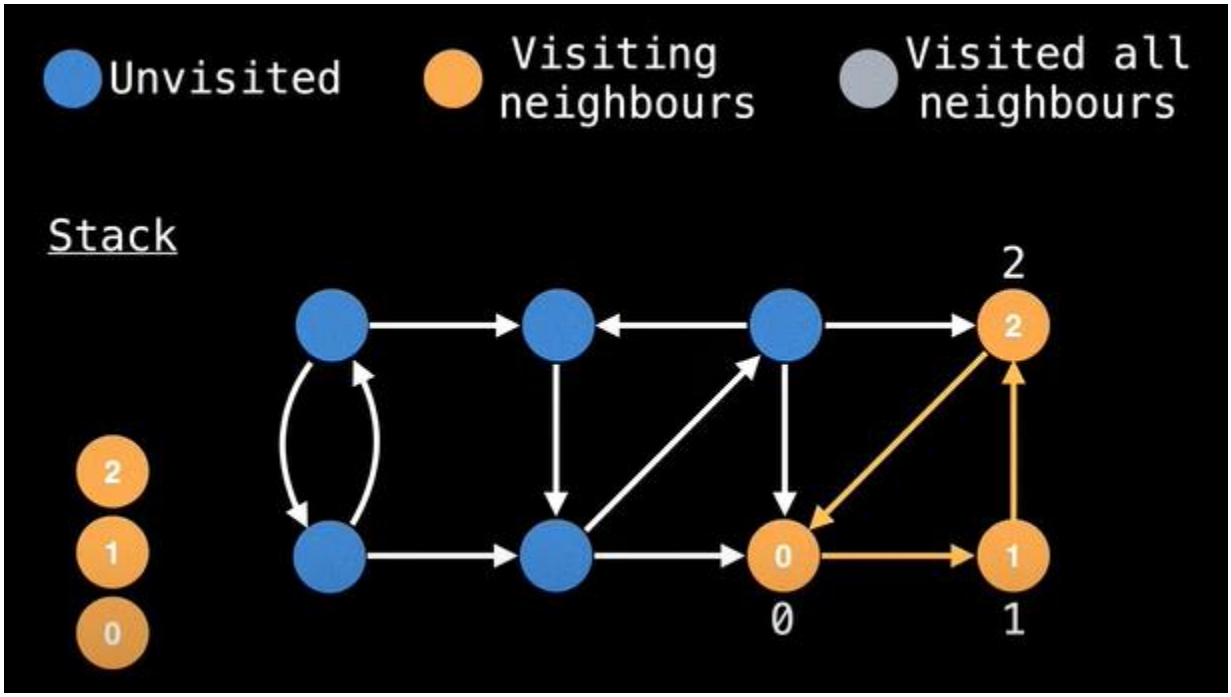
بنابراین نیازمند تغییر در روش و اضافه کردن شروطی به آن هستیم.

در روش جدید، از استک استفاده میکنیم و سه مجموعه Unvisited, Visiting neighbours, Visited all neighbours را با رنگ های متمایز در نظر میگیریم:



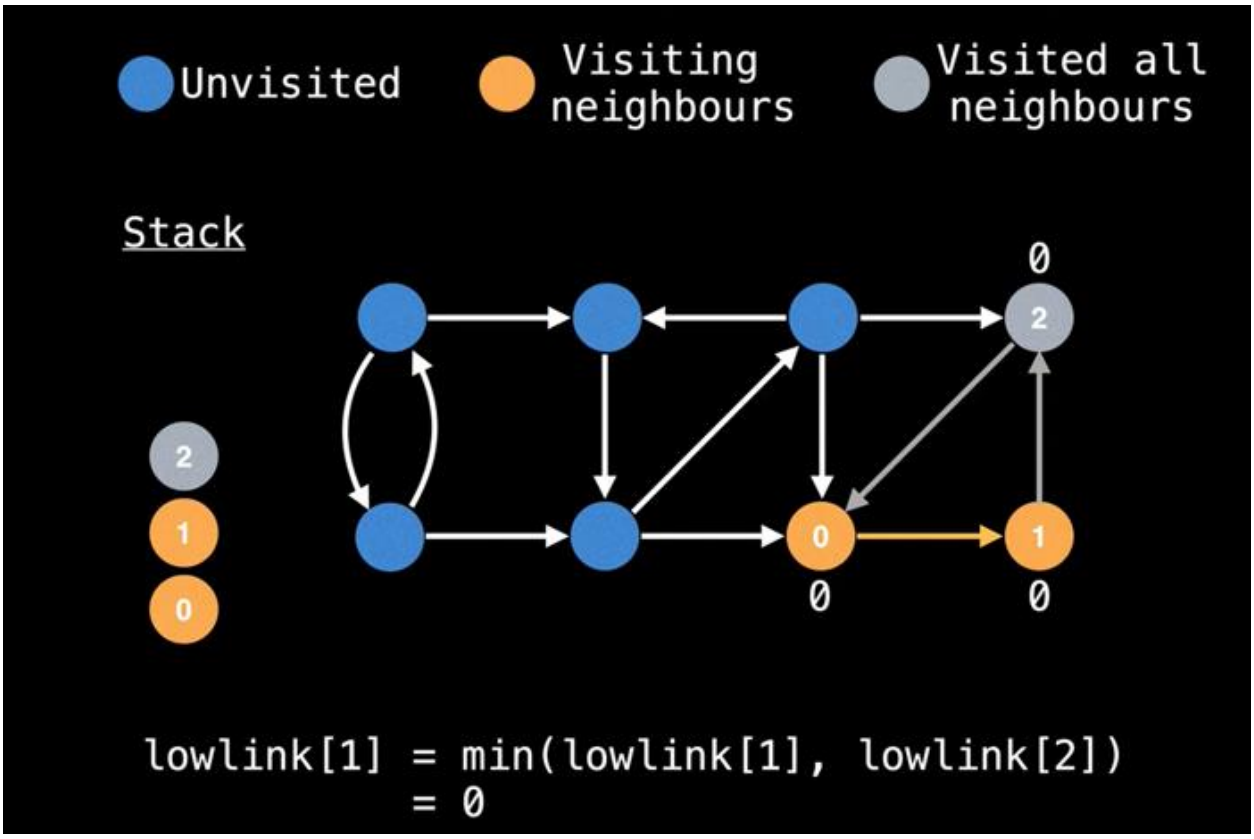
با شروع از راس دلخواه، هر راسی را که میبینم در استک push میکنیم و متغیر بولین visited آن را برابر با true قرار میدهیم. متغیر visited نشان دهنده قرار داشتن یک راس در استک است.

همچنین به هر راس یک id میدهیم و Low-Link Value آن را نیز برابر همان id قرار میدهیم:

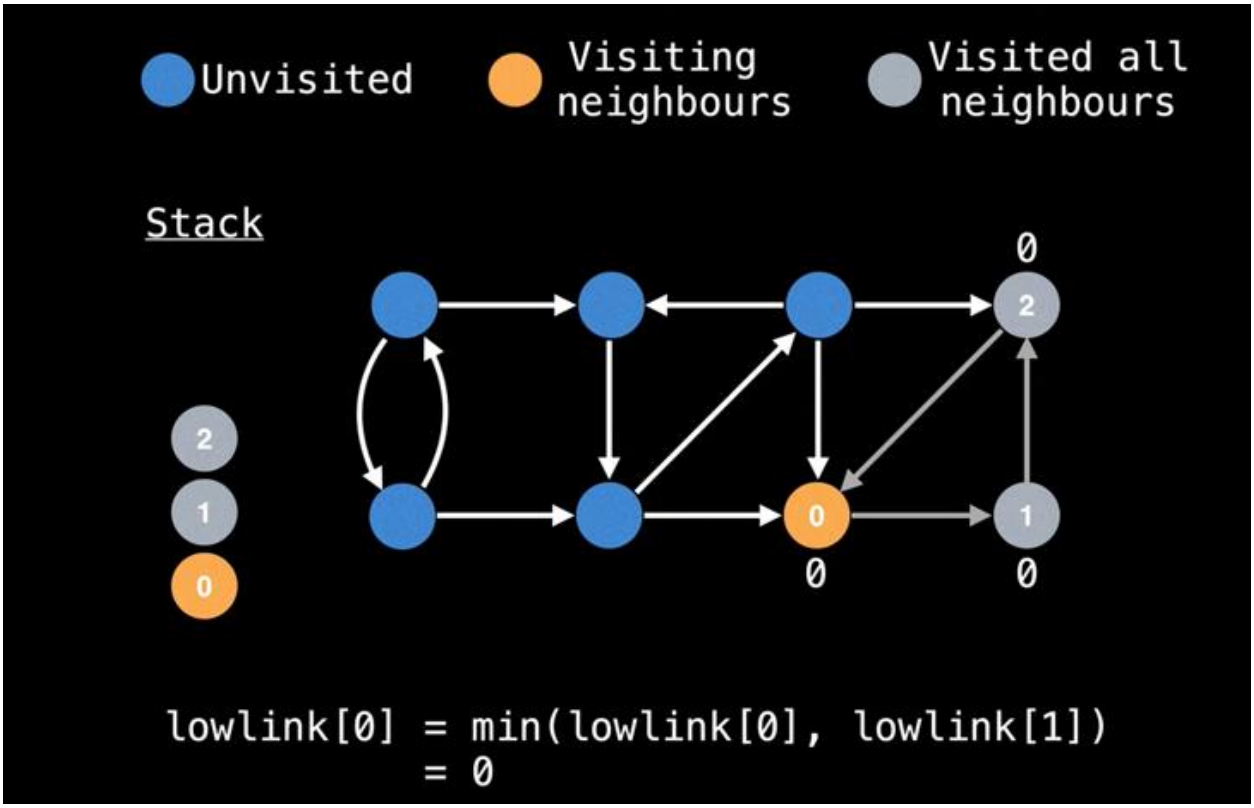


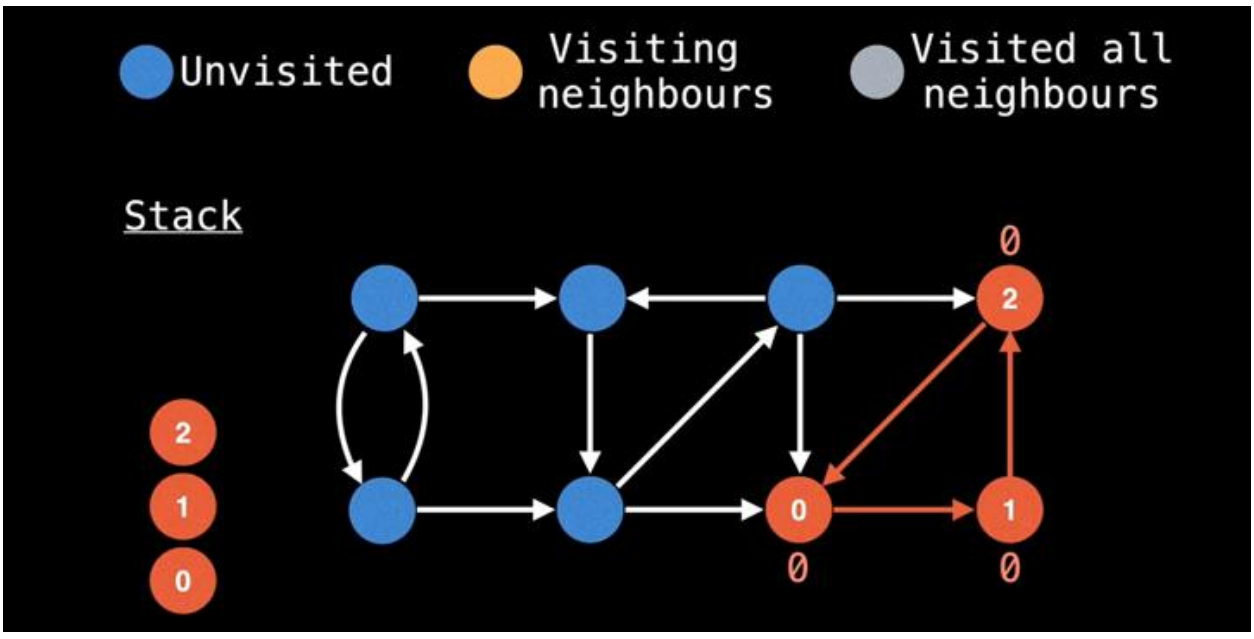
هنگام پیشمایش راس ها، اگر متغیر visited راسی true بود، یعنی قبلا در این پیمایش دیده شده و نیازی به دیدن مجدد آن و قرار دادن آن در استک نداریم.

هنگامی که به چنین راسی برخورد کردیم، راسی که قبل از آن دیده بودیم را به رنگ خاکستری در می آوریم و Low-Link Value آن را برابر با مینیمم Low-Link Value دو راس قرار میدهیم:



این کار را برای همه راس های نارنجی دیگر تکرار میکنیم تا زمانی که همه راس ها خاکستری شده و Low-Link Value درست برای آن مشخص شود:

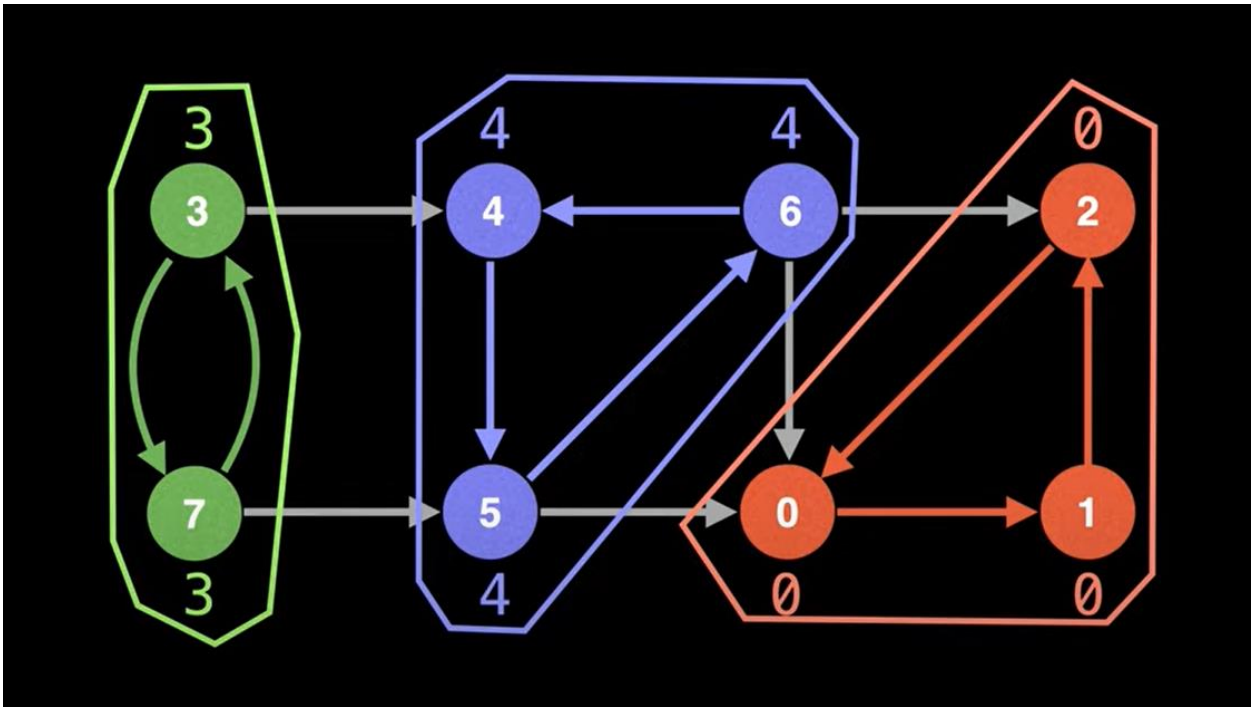




پس از اینکه همه راس های نارنجی به خاکستری تبدیل شدند، راس ها را از استک pop میکنیم.

سپس از یک راس دیگر Unvisited در گراف که به رنگ آبی است شروع به پیمایش میکنیم و مراحل گفته شده را انجام میدهیم.

پس از اینکه مراحل را برای همه راس های گراف طی کردیم، به شکل زیر میرسیم:



همانطور که مشخص است، توانستیم با روش استفاده از استک، با شروع از راس دلخواه، تمام SSC های گراف را به درستی مشخص کنیم.

لازم به ذکر است پیچیدگی زمانی الگوریتم Tarjan، O(V + E) است. V تعداد راس ها و E تعداد یال هاست.

در واقع پیچیدگی زمانی این الگوریتم، مشابه پیچیدگی زمانی DFS است.

چرا که در هر بار پیمایش عمقی که انجام میدهیم، همه راس ها را بررسی نمیکنیم و جایی که به راس تکراری (قبلا دیده شده) برخورد کنیم، متوقف میشویم و SCC را تشکیل میدهیم.

همچنین مقدار حافظه مورد نیاز این الگوریتم O (V + E) است.

با توجه به اینکه از از استک و حافظه های جانبی استفاده شده، مقدار حافظه ی استفاده شده در این الگوریتم کمی بیشتر از V + E است اما از نظر اردر همان O(V + E) است.

فایل detect_cycle.c، الگوریتم Tarjan توضیح داده شده را پیاده سازی میکند:

```
#define UNVISITED -1
bool *visited;
int *ids, low, sccs;
Stack stack;
int id;
int loop_index = 0;

void dfs(int at, list *loops, bool hasloop, node **adjacencyList)
{
    low[at] = id;
    ids[at] = id;
    id++;
    stack.push(at);
    visited[at] = true;
    for (int i = 0; i < sizeof(adjacencyList[at] / sizeof(adjacencyList[at][0])); i++)
    {
        if (ids[i] == UNVISITED)
        {
            dfs(i, loops, &hasloop, adjacencyList);
        }
        if (visited[i])
        {
            low[at] = min(low[at], low[i]);
        }
    }
```

```
    if (ids[at] == low[at])
    {
        while (0 < stack->size)
        {
            int n = stack.pop();
            visited[n] = false;
            if (adjacencyList[n][0]->type == Thread)
            {
                push(loops[loop_index], adjacencyList[n][0]->id);
            }
            if (n == at)
            {
                if (loops[loop_index]->size == 1)
                {
                    pop(loops[loop_index]);
                    break;
                }
                hasloop = true;
                loop_index++;
                break;
            }
        }
    }
    return;
}
```

```
52  void tarjan(Graph *graph)
53  {
54      int loop_index = 0;
55      bool hasLoop = graph->hasloop;
56      list *loops = graph->loops;
57      node **adjList = graph->adjacencyList;
58      visited = (int *)malloc(length);
59      low = (int *)malloc(length);
60      sccs = (int *)malloc(length);
61      ids = (int *)malloc(length);
62      *visited = {false};
63      *ids = {UNVISITED};
64      id = 0;
65      for (int i = 0; i < length; i++)
66      {
67          if (ids[i] == UNVISITED)
68          {
69              dfs(i, loops, hasLoop, adjList);
70          }
71      }
72  }
```

ابتدا متغیر های مورد نیاز را به صورت global تعریف میکنیم.

ماکرو UNVISITED با استفاده از دستور #define برابر 1- تعریف شده است.

سپس آرایه های visited, ids, low, sccs تعریف شده اند.

سپس یک استک تعریف شده است.

متغیر های id و loop_index نیز تعریف شده اند که loop_index نشان دهنده تعداد حلقه ها (SSC) های موجود در گراف است.

پس از تعریف متغیرها، تابع dfs را داریم.

ورودی های این تابع به شرح زیر هستند:

int at، راسی که در حال حاضر در حال پیماش آن هستیم را نشان میدهد.

list* loops، آرایه ای از لیست هاست به صورتی که هر لیست آن، یک حلقه را نگه میدارد.

bool hasloop، نشان دهنده این است که آیا در کل گراف حلقه داریم یا خیر.

node **adjacencyList، ماتریس مجاورت ماست.

حال به بدنه تابع میپردازیم.

ابتدا مقادیر low[at], ids[at] برابر با at میشوند.

یعنی id و Low-link Value برای راسی که در حال پیمایش آن هستیم مقداردهی میشود.

سپس به مقدار id یکی اضافه میکنیم.

at را در استک push میکنیم و visited[at] را true میکنیم.

سپس در حلقه for، در هر مرحله چک میکنیم که اگر یک راس دیده نشده بود (ids[i] == UNVISITED)، تابع را به صورت بازگشتی صدا میزنیم.

و در صورت true بودن مقدار visited[i]، low[at] (Low-link Value) را برابر مینیمم low[i] , low[at] قرار میدهیم (در قسمت توضیحات الگوریتم در رابطه با دلیل انجام این کار توضیح داده شده است).

سپس چک میکنیم که اگر مقادیر low[at], low[i] با هم برابر بودند (یعنی یک SCC را تشخیص دادیم و پیمایشمان به پایان رسیده است)، تا زمانی که استک خالی نشده است، عناصر استک را pop میکنیم و visited آنها را false میکنیم.

راس pop شده را در لیستی که با مقدار loop_index به آن اشاره میشود، ذخیره میکنیم.

نکته مهم این است که باید راسی که ذخیره میشود، حتما از جنس Thread باشد. چرا که ما در بخش recovery نیاز به Thread داریم و به resource نیازی نداریم.

در قسمت بعد چک میکنیم که اگر n (عنصر pop شده از استک) با at برابر بود، در صورتی که سایز حلقه loop_index ام برابر با یک بود (یعنی طوقه بود)، آن حلقه را از آرایه حلقه ها حذف کرده و break میکنیم.

دلیل این کار این است که تشخیص طوقه برای ما فایده ای ندارد. چون هر راس یک thread است و یال از خودش به خودش بی معناست.

پس از این شرط و در صورتی که حلقه ما طوقه نبود، مقدار has_loop را true میکنیم و loop_index را یکی اضافه میکنیم. در واقع میگوییم در گراف حلقه وجود دارد و یکی به تعداد حلقه ها اضافه میکنیم.

در بخش آخر، تابع tarjan را داریم که در ورودی خود یک graph میگیرد.

مقدار دهی های اولیه را با استفاده از دستوراتی مانند malloc انجام میدهیم و همچنین در ابتدا visited و ids را برای همه عناصر آرایه برابر false و UNVISITED قرار میدهیم.

سپس id را برابر صفر قرار میدهیم.

در انتها با استفاده از یک حلقه، کل گراف را پیمایش میکنیم و در صورتی که هر راس دیده نشده بود، تابع dfs را برای آن صدا میزنیم.

حال ما باید تغییراتی را که لازم است در سیستم عامل pintos ایجاد کنیم.

در ابتدا در بخش شروع سیستم عامل تابع آماده سازی، فرایند شناسایی و بازیابی را شروع می کنیم :

```
main (void)
{


  ////////////////////OUR_CHANGE-----------------------------------------------------------------------
  init_deadlock_detection();
  ////////////////////OUR_CHANGE-----------------------------------------------------------------------


}
```

سپس وارد بخش تایمر سیستم عامل می شویم و در آن تابعی را که به وسیله آن بن بست را در کد پیدا میکنیم و آن بن بست را به شیوه ای که در آینده توضیح می دهیم، هندل می کنیم، هر چند کلاک یکبار صدا میزنیم.(در این جا هر 100 تیک)

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  enum intr_level old_level = intr_disable ();

  wake_up_sleeping_threads ();
  thread_tick ();

  intr_set_level (old_level);

  ////////////////////OUR_CHANGE-----------------------------------------------------------------------
  int fullInterval = 100;
  if((ticks % fullInterval) == 0){
      deadlock_detection_recovery();
  }
  ////////////////////OUR_CHANGE-----------------------------------------------------------------------

}
```

حال ما باید در فایل synch.h در استراکت قفل مان تغییری ایجاد کنیم. این تغییر تنها اضافه کردن یک آیدی به قفل مان می باشد:

```
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
    struct list_elem lock_elem; /* List elem of lock from structure thread */
    int lock_priority;          /* The highest priority waiting for the lock. */

    ////////////////////OUR_CHANGE-----------------------------------------------------
    int lock_id;
    ////////////////////OUR_CHANGE-----------------------------------------------------
  };
```

حال در تابع lock_acquire ما باید درخواست منابع برای یک thread خاص را به سیستم عامل اعلام کنیم. بعد از اینکه سمافور به ما اجازه داد تا منابع در اختیار بگیریم تابع گرفتن منابع را صدا میزنیم(در واقع این دو تابع مستقیم با سیستم عامل کاری ندارند بلکه تنها آن thread را به قفلی که داریم در گرافی در آینده توضیح خواهیم داد ثبت می‌کنیم تا به هنگام بررسی و بازیابی بن بست از آن استفاده کنیم.

```
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));
  enum intr_level old_level;
  old_level = intr_disable ();
  struct thread *curr, *thread;
  struct lock *max_lock;
  curr = thread_current();
  thread = lock->holder;
  curr->blocked = max_lock = lock;
  if (!thread_mlfqs)
  {
    while (thread != NULL && thread->priority < curr->priority)
      {
        thread->donated = true;
        thread_set_priority_extra (thread, curr->priority, false);
        if (max_lock->lock_priority < curr->priority){
          max_lock->lock_priority = curr->priority;
        }
        if (thread->status == THREAD_BLOCKED && thread->blocked != NULL)
          {
            max_lock = thread->blocked;
            thread = thread->blocked->holder;
          }
        else
          break;
      }
  }
  ////////////////////OUR_CHANGE-----------------------------------------------
  request_resources(curr->tid , lock->lock_id);
  sema_down (&lock->semaphore);
  receives_resources(curr->tid , lock->lock_id);
  ////////////////////OUR_CHANGE-----------------------------------------------
  lock->holder = curr;
  curr->blocked = NULL;

  if (!thread_mlfqs)
    list_push_back(&lock->holder->locks, &lock->lock_elem);
  intr_set_level(old_level);
}
```

بعد از پایان کارمان ما باید آن قفل را آزاد کنیم پس همانطور که در تابع lock_release می بینید، به محض اینکه سمافور به ما اجازه دهد ما منابعی که مربوط است به آن thread و آن قفل را آزاد می کنیم (در واقع در گراف مان آن را حذف می‌کنیم).

```
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  struct thread *curr;
  struct list_elem *elem;
  struct lock* max_lock;
  enum intr_level old_level;

  old_level = intr_disable();
  curr = thread_current ();

  lock->holder = NULL;
  sema_up (&lock->semaphore);
  ////////////////////OUR_CHANGE-----------------------------------------------------------------------
  fress_resource(curr->tid , lock->lock_id);
  ////////////////////OUR_CHANGE-----------------------------------------------------------------------
  if (!thread_mlfqs)
  {
    list_remove (&lock->lock_elem);
    lock->lock_priority = PRI_MIN - 1;

    if (list_empty(&curr->locks))
      {
        curr->donated = false;
        thread_set_priority (curr->base_priority);
      }
    else
      {
        elem = list_max (&curr->locks, &lock_less_func, NULL);
        max_lock = list_entry (elem, struct lock, lock_elem);
        if (max_lock->lock_priority != PRI_MIN - 1)
          thread_set_priority_extra (curr, max_lock->lock_priority, false);
        else
          thread_set_priority (curr->base_priority);
      }
  }

  intr_set_level (old_level);
}
```

حال ما باید تغییری در بخش thread مان نیز ایجاد کنیم. چون در آینده به علت کمبود منابع یا برخورد با بن‌بست مجبوریم بعضی از thread هایمان را قطع کنیم به همین علت یک شمارنده برای هر thread می گذاریم تا بدانیم یک thread چند بار قربانی شده است.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    ////////////////////OUR_CHANGE-------------------------------------------------------------------------
    int victimed_count;
    ////////////////////OUR_CHANGE-------------------------------------------------------------------------
```

درنهایت نیز باید در فایل process.c به هنگام خارج شدن از یک پراسس thread مد نظرمان را به آن بدهیم:

```
////////////////////OUR_CHANGE-------------------------------------------------------------------------
void
process_exit (struct thread *cur)
{
  //.................................
```

تابع هایی که در بالا استفاده شدند، در فایل recovery.c قرار دارند که به شرح زیر می باشند:

```
static void init_deadlock_detection(){
    graph = create_graph()
}
static void request_resources(int tid , int lock_id){
    add_edge(graph , tid , lock_id , resource_type)
}
static void receives_resources(int tid , int lock_id){
    delete_edge(graph, tid ,lock_id);
    add_edge(graph , tid , lock_id , thread_type)
}
static void frees_resource(int tid , int lock_id){
    delete_edge(graph, tid ,lock_id);
}
```

برای ایجاد این توابع ساختارهایی استفاده کردیم که به شرح زیر میباشند:

```
#define N 10000

typedef enum{
    thread , resource
} Type;

struct Node{
    int id;
    Type type;
    struct Node* next;
}

struct list
{
  int data;
  struct list *next;
};

void push(struct list** head_ref, int new_data);

struct Graph{
    struct Node* adjacency_lists[N];
    bool hasLoop;
    struct list* loops;
}

struct Graph* createGraph();
void add_edge(struct Graph* graph, int tid , int lock_id, enum Type type);
void delete_edge(struct Graph* graph, int tid , int lock_id);
```

```c
#define N 10000

void push(struct list** head_ref, int new_data)
{
    struct list* new_list = (struct list*) malloc(sizeof(struct list));
    new_list->data  = new_data;
    new_list->next = (*head_ref);
    (*head_ref)    = new_list;
}

struct Graph* createGraph()
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    for (int i = 0; i < N; i++) {
        graph->adjacency_lists[i] = (struct Node*)malloc(sizeof(struct Node));
    }
    return graph;
}

void add_edge(struct Graph* graph, int tid , int lock_id, enum Type type)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->id = lock_id;
    newNode->type = type;
    newNode->next = graph->adjacency_lists[tid];
    graph->adjacency_lists[tid] = newNode;
}

void delete_edge(struct Graph* graph, int tid , int lock_id)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    graph->adjacency_lists[tid] = newNode;
}
```

در بخش آخر، به نحوه ی recovery کردن از Deadlock میپردازیم.

```
11  #define get_length(arr) (sizeof((arr)) / sizeof(arr[0]))
12  struct Graph *graph;
13
14  enum Type resource_type = resource;
15  enum Type thread_type = thread;
```

```
83
84  static void init_deadlock_detection(){
85      graph = create_graph()
86  }
87  static void request_resources(int tid , int lock_id){
88      add_edge(graph , tid , lock_id , resource_type)
89  }
90  static void receives_resources(int tid , int lock_id){
91      delete_edge(graph, tid ,lock_id);
92      add_edge(graph , tid , lock_id , thread_type)
93  }
94  static void frees_resource(int tid , int lock_id){
95      delete_edge(graph, tid ,lock_id);
96  }
97
```

در ابتدا تابع **init_deadlock_detection** را داریم که در main برنامه اصلی فراخوانی میشود و گراف اولیه ما را میسازد.

تابع **request_resource** برای زمانی است که یک thread سعی میکند که یک lock را acquire کند و با این کار یک یال از thread به resource اضافه میشود به گرافمان.

تابع **receives_resource** موقعی فراخوانی می شود که میخواهیم lock را دریافت کنیم و باید جهت edge را در گراف برعکس کنیم تا تغییر حالت را نشان دهیم.

تابع **frees_resource** وقتی که Thread به عنوان victim برای kill شدن انتخاب میشود، باید resource های آن را آزاد کند و یالی که در گراف ایجاد شده است را پاک کند.

```
68  }
69  static bool detect_deadlock()
70  {
71      tarjan(&graph); //detect loops
72      return graph->hasCycle;
73  }
74
75  static void deadlock_detection_recovery()
76  {
77      bool deadlock = detect_deadlock();
78      if (deadlock)
79      {
80          recover(graph->loops);
81      }
82  }
```

قبل از اینکه بخواهیم به فکر recovery باشیم باید اول مطمئن شویم deadlock رخ داده است. برای این کار تابعی به نام detect_deadlock تعریف می کنیم که در آن تابع الگوریتم detection را صدا میزند.

همانطور که در بخش های قبلی بررسی شده است الگوریتم detection ما tarjan است و تابع مربوطه با ورودی گرفتن گراف کار خود را شروع میکند و پس از آن اگر graph حداقل یک حلقه را تشخیص دهد، متغیر hasloop را مقدار دهی میکنیم.

اگر این متغیر مقدار true داشته باشد پس deadlock رخ داده است و با فراخوانی تابع recover سیستم را از حالت deadlock خارج میکنیم.

```
55  static void recover(list* cycles)
56  {
57      for(int i = 0 ; i < get_length(cycles) ; i++ )
58      {
59          struct thread *victim = choose_victim(cycles[i]);
60          victim->victimed_count = victim->victimed_count++;
61          while (!list_empty(&victim->locks))
62          {
63              struct lock *lock = list_pop_front(&victim->locks);
64              lock_release(lock, victim);
65          }
66          process_exit(victim);
67      }
```

تابع **recover** روی یک آرایه از list ها که در هر list شناسه thread هایی که در دور دخیل هستند ذخیره شده است، اجرا میشود.

بر این اساس با iterate کردن روی آرایه از cycle ها و انتخاب کردن یک victim از هر کدام از دور ها سعی میکنیم deadlock رخ داده را از بین ببریم.

```
18  static struct thread *choose_victim(list* cycle)
19  {
20      int average = 0;
21      srand(time(0));
22      int length = get_length(cycle);
23      for (int i = 0; i < length; i++)
24      {
25          average += thread_get_by_id(cycle[i])->victimed_count;
26      }
27      int *low_victims = (int *)malloc(length * sizeof(int));
28      int *high_victims = = (int *)malloc(length * size(int));
29      int hight_count = 0;
30      int low_count = 0;
31      for (int i = 0; i < length; i++)
32      {
33          if (thread_get_by_id(cycle[i])->victimed_count > average)
34          {
35              high_victims[hight_count++] = cycle[i];
36          }
37          else
38          {
39              low_victims[low_count++] = cycle[i];
40          }
41      }
42      struct thread *victim ;
43      int rand_num = rand()% 9 ;
44      if(rand_num > 2 || get_length(high_victims) == 0){
45          int rand_victim = rand() % low_count ;
46          victim = thread_get_by_id(low_victims[rand_victim]);
47      } else{
48          int rand_victim = rand() % hight_count ;
49          victim = thread_get_by_id(high_victims[rand_victim]);
50      }
51      free(low_victims);
52      free(high_victims);
53      return victim;
54  }
```

در تابع **choose_vicitm** با گرفتن یک list از thread id ها که نمایانگر node های دور هستند، با میانگین گیری روی مقدار متغیر victimed_count سعی میکنیم thread هایمان را به دو گروه high و low تقسیم کنیم پس از آن از بین این دو گروه با احتمال 3(high) به 6(low) به صورت random یکی را انتخاب کنیم تا kill شود.

دلیل برای این روش از پیاده سازی برای این است که thread هایی که کمتر victim شده اند را با احتمال بیشتری انتخاب کنیم در عین حال شانس انتخاب شدن برای تمام thread ها موجود باشد.

از نظر زمانی میتوان ثابت کرد که این تابع در order زمانی n(اندازه آرایه ورودی) عمل میکند به دلیل اینکه هر یک از عناصر آرایه در یک loop خطی بررسی میشود. اگر دقیق تر بخواهیم درباره order زمانی صحبت کنیم به اندازه N*2 است تقریبا به دلیل وجود دو حلقه.

اما نکته مهم تر در زمان اجرای خود تابع recover دیده میشود. چون در داخل آن تابع یک حلقه بر روی همه دور ها داریم ولی چون با استفاده از الگوریتم trajan به این رسیده ایم میدانیم که یک node فقط میتواند در یک حلقه موجود باشد (تعریف SCC) پس در بدترین حالت اگر تمامی node های گراف در دور دخیل باشند یعنی اندازه آرایه مان V میباشد و تعداد SCC ها باید همیشه کوچکتر یا مساوی با تعداد گره های گراف باشد میتوانیم ببینیم که زمان اجرا recovery از deadlock در زمان خطی برابر با تعداد گره هایمان در بدترین حالت صورت میگیرد.

با تشکر از زحمات و همراهی جناب دکتر شهاب الدین نبوی