

فهرست

۳.....	مقدمه	۱
۴.....	تمرینات	۲
۴.....	تمرین ۱ - الگوریتم FCFS	۲.۱
۵.....	تمرین ۲ - الگوریتم اولویت زمان‌بندی	۲.۲
۶.....	تمرین ۳ - الگوریتم اولویت	۲.۳
۷.....	تمرین ۴ - الگوریتم پترسون	۲.۴
۱۰.....	تمرین ۵ - الگوریتم نانوایی	۲.۵
۱۳.....	تمرین ۶ - مسئله تولیدکننده مصرف‌کننده و شام فیلسفان	۲.۶
۱۸.....	تمرین ۷ - مسئله خوانندگان و نویسنده‌گان	۲.۷
۲۰.....	تمرین ۸ - مقاله	۲.۸
۲۵.....	تمرین اضافه - سیستم عامل‌های بلاذرنگ	۲.۹
۲۸.....	تمرین اضافه - پیدا کردن حفره‌های متوالی	۲.۱۰
۳۰.....	تمرین ۹ - الگوریتم جایگزینی	۲.۱۱
۳۵.....	تمرین اضافه - دسترسی فایل در لینوکس	۲.۱۲
۳۶.....	تمرین ۱۰ - بردار وقفه و DMA	۲.۱۳
۳۸.....	منابع	۳

۱ مقدمه

در دنیای پر شتاب فناوری اطلاعات، سیستم‌های عامل نقش محوری در مدیریت منابع کامپیوتری ایفا می‌کنند. یکی از مهم‌ترین وظایف سیستم عامل، مدیریت فرآیندها و تخصیص منابع به آن‌ها است. در این گزارش کار، به بررسی و تحلیل الگوریتم‌های مختلف زمان‌بندی فرآیندها و مکانیزم‌های همگام‌سازی پرداخته شده است. هدف اصلی، درک عمیق از نحوه عملکرد این الگوریتم‌ها و تأثیر آن‌ها بر کارایی سیستم است.

در کل، به صورت عملی به پیاده‌سازی و شبیه‌سازی الگوریتم‌های زمان‌بندی مانند FCFS و صف اولویت پرداخته شده است. همچنین، مسائل همگام‌سازی پیچیده‌تری مانند الگوریتم پترسون، نانوایی، تولید‌کننده-صرف‌کننده و غذاخوردن فیلسوفان مورد بررسی قرار گرفته است. با استفاده از ابزارهای شبیه‌سازی و کامپایلر، عملکرد این الگوریتم‌ها در سناریوهای مختلف ارزیابی شده و نتایج حاصل تحلیل شده است.

نتایج حاصل نشان می‌دهد که انتخاب الگوریتم زمان‌بندی مناسب و استفاده از مکانیزم‌های همگام‌سازی کارآمد، تأثیر بسزایی در عملکرد کلی سیستم دارد. همچنین، با مقایسه نتایج حاصل از شبیه‌سازی با مطالعات قبلی، به درک بهتری از مزایا و معایب هر یک از الگوریتم‌ها دست یافته‌ایم. در نهایت، با توجه به نتایج بدست آمده، پیشنهاداتی برای بهبود عملکرد سیستم‌های عامل ارائه شده است.

۲ تمرینات

۲/۱ تمرین ۱ - الگوریتم FCFS

میانگین زمان پاسخ با استفاده از الگوریتم زمانبندی FCFS برای پنج پروسه P1 تا P5 با زمانهای ورود به ترتیب ۰، ۱۰، ۴۰، ۵۰، ۵۰ و زمانهای اجرای ۳۰، ۲۰، ۵۰، ۱۰ و ۳۰ میلی ثانیه با در نظر گرفتن زمان تعویض متن ۲ ثانیه را محاسبه نمایید.

۲/۱/۱ درک مسئله

ما در این مسئله با پنج فرآیند روبرو هستیم که هر کدام زمان ورود و زمان اجرای مشخصی دارند. هدف ما محاسبه میانگین زمان پاسخ با استفاده از الگوریتم FCFS است. همچنین باید تأثیر زمان تعویض متن را نیز در نظر بگیریم.

۲/۱/۲ الگوریتم FCFS

در الگوریتم FCFS، فرآیندی که زودتر وارد سیستم شود، زودتر نیز اجرا می‌شود. به عبارت دیگر، فرآیندها به ترتیب ورود به صفت‌پردازش قرار می‌گیرند.

۲/۱/۳ مراحل حل مسئله

- ایجاد جدول: جدولی ایجاد می‌کنیم که در آن برای هر فرآیند، زمان ورود، زمان اجرا، زمان شروع اجرا، زمان پایان اجرا و زمان پاسخ را ثبت می‌کنیم.
- نمودار گانت: نموداری ترسیم می‌کنیم که در آن محور افقی زمان و محور عمودی فرآیندها را نشان می‌دهد. هر فرآیند با یک مستطیل نشان داده می‌شود که طول آن برابر با زمان اجرای فرآیند و موقعیت آن روی محور افقی نشان‌دهنده زمان شروع اجراست.
- محاسبه زمان شروع و پایان اجرا: با توجه به الگوریتم FCFS، فرآیندها به ترتیب ورود اجرا می‌شوند. زمان شروع اجرای هر فرآیند برابر با زمان پایان اجرای فرآیند قبلی به علاوه زمان تعویض متن است. زمان پایان اجرا نیز برابر با زمان شروع اجرا به علاوه زمان اجرای فرآیند است.
- محاسبه زمان پاسخ: زمان پاسخ برای هر فرآیند برابر است با تفاضل زمان پایان اجرا و زمان ورود فرآیند.
- محاسبه میانگین زمان پاسخ: میانگین زمان پاسخ را با تقسیم مجموع زمان‌های پاسخ همه فرآیندها بر تعداد فرآیندها محاسبه می‌کنیم.

حل مسئله در شکل زیر آمده:

	Year	Month	Week	Day	Hour	Minute	Second	Time	Process	Arrival Time	Start Time	Completion Time	Turnaround Time	Waiting Time	Preemptive
									P ₁	0	30	30	30	0	0
									P ₂	10	20	52	52	32	32
									P ₃	40	50	104	104	52	54
									P ₄	50	10	116	116	104	106
									P ₅	60	30	148	148	116	148
															FCFS

تمرین ۱:

$AT = \frac{30 + 152 + 104 + 116 + 290}{5} = 53$

شکل ۱ - تمرین ۱ // الگوریتم FCFS

۲/۲ تمرین ۲ - الگوریتم اولویت زمان‌بندی

میانگین زمان انتظار با استفاده از الگوریتم زمان‌بندی اولویت برای پنج پروسه با زمانهای ورود صفر و زمانهای اجرای به ترتیب ۹، ۶، ۱، ۵ و ۴ و اولویت ۴، ۱، ۲، ۳ و ۲ را تعیین نمایید.

۲/۲/۱ درک مسئله

در این مسئله، با پنج فرآیند روبرو هستیم که همگی در زمان صفر وارد سیستم می‌شوند. اما هر فرآیند اولویت مشخصی دارد. هدف ما محاسبه میانگین زمان انتظار با استفاده از الگوریتم اولویت است. در این الگوریتم، فرآیندی که اولویت بالاتری دارد، زودتر اجرا می‌شود.

۲/۲/۲ مراحل حل مسئله

- مرتب‌سازی فرآیندها: فرآیندها را بر اساس اولویت به صورت نزولی مرتب می‌کنیم. فرآیند با بالاترین اولویت ابتدا اجرا می‌شود.
- محاسبه زمان شروع و پایان اجرا: با توجه به ترتیب اولویت، زمان شروع و پایان اجرای هر فرآیند را محاسبه می‌کنیم. زمان شروع اجرای هر فرآیند برابر با زمان پایان اجرای فرآیند قبلی است.
- محاسبه زمان انتظار: زمان انتظار برای هر فرآیند برابر است با تفاضل زمان شروع اجرا و زمان ورود فرآیند (که در این مثال برای همه فرآیندها صفر است).
- محاسبه میانگین زمان انتظار: میانگین زمان انتظار را با تقسیم مجموع زمان‌های انتظار همه فرآیندها بر تعداد فرآیندها محاسبه می‌کنیم.

حل مسئله در شکل زیر آمده:

	ورود پرسه	اعیا	لاریج	شروع	۹	۱۳	۱۴	۱۷	۲۵	۲۵
P ₁	0	9	4	9	P ₁	P ₅	P ₃	P ₄	P ₂	
P ₂	0	6	1	25	ATT = (9-0) + (25-9) + (14-1) + (19-14) + (13-19)					5
P ₃	0	1	2	14						
P ₄	0	5	2	19	ATT = $\frac{92}{5} = 18.4$					
P ₅	0	4	3	13	AWT = $\frac{(9-7)+(25-6)+(14-1)+(19-5)+(13-4)}{5} = \frac{55}{5} = 11$					
عملیات اولویت سیر										

شکل ۲ - تمرین ۲ حل به روش صفت اولویت انحصاری

۲/۳ تمرین ۳ - الگوریتم اولویت

چهار پروسه ۱ تا P4 با زمانهای ورود به ترتیب ۰، ۱، ۲ و ۳ و زمانهای سرویس ۴، ۳، ۷ و ۹ و اولویت ۲، ۳، ۱ و ۴ در نظر بگیرید با استفاده از زمانبند اولویت قابل پس گرفتن، متوسط زمان انتظار را محاسبه نمایید.

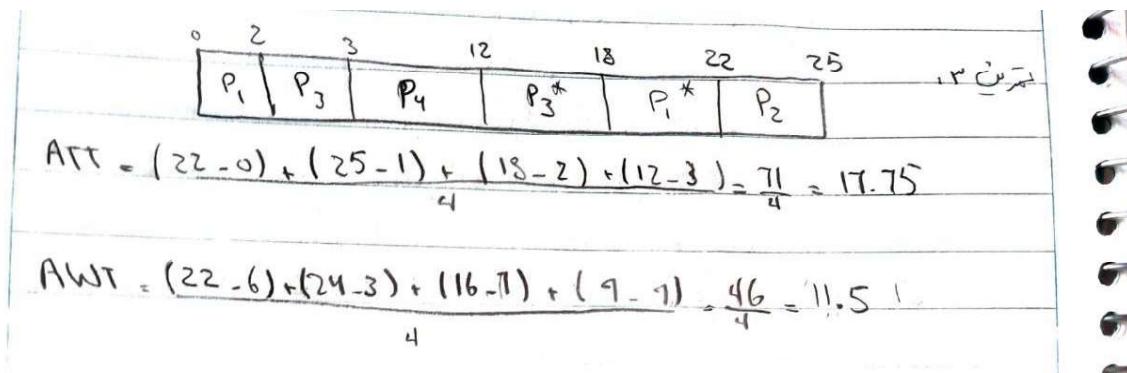
۲/۳/۱ درک مسئله

در این مسئله با چهار فرآیند رویرو هستیم که هر کدام زمان ورود، زمان سرویس (زمان اجرا) و اولویت مشخصی دارند. از آنجایی که از زمانبند اولویت قابل پس گرفتن استفاده می‌شود، هر زمان فرآیندی با اولویت بالاتر وارد سیستم شود، فرآیند در حال اجرا متوقف شده و فرآیند جدید شروع به اجرا می‌کند.

۲/۳/۲ مراحل حل مسئله

- مرتب‌سازی بر اساس زمان ورود: در ابتدا فرآیندها را بر اساس زمان ورود مرتب می‌کنیم تا ترتیب ورود آن‌ها به سیستم مشخص شود.
- جدول زمانبندی: جدولی ایجاد می‌کنیم که در آن زمان، فرآیند در حال اجرا و سایر اطلاعات مورد نیاز را ثبت می‌کنیم.
- شبیه‌سازی: در نمودار گانت با شروع از زمان صفر، فرآیندها را به ترتیب ورود بررسی می‌کنیم. اگر فرآیند جدیدی با اولویت بالاتر از فرآیند در حال اجرا وارد شود، فرآیند در حال اجرا متوقف شده و فرآیند جدید شروع به اجرا می‌کند. این کار تا زمانی ادامه می‌یابد که همه فرآیندها به پایان برسند.
- محاسبه زمان انتظار: زمان انتظار برای هر فرآیند برابر است با تفاضل زمان شروع اجرای واقعی و زمان ورود فرآیند.
- محاسبه میانگین زمان انتظار: میانگین زمان انتظار را با تقسیم مجموع زمان‌های انتظار همه فرآیندها بر تعداد فرآیندها محاسبه می‌کنیم.

حل مسئله در شکل زیر آمده:



شکل ۳- تمرین ۳ روش اولویت غیر انحصاری

۲/۴ تمرین ۴ - الگوریتم پترسون

الگوریتم Peterson برای N پروسه را به طور کامل تشریح و پیاده سازی نمایید.

۲/۴/۱ الگوریتم پترسون برای N پروسه

الگوریتم پترسون به عنوان یک راه حل کلاسیک برای حل مسئله انحصار متقابل (Mutual Exclusion) در سیستم‌های همرونده شناخته می‌شود. این الگوریتم به دو فرآیند اجازه می‌دهد تا به صورت ایمن از یک منبع مشترک استفاده کنند بدون اینکه به داده‌های مشترک آسیب برسد. گرچه الگوریتم اصلی پترسون برای دو فرآیند طراحی شده است، اما می‌توان آن را برای N فرآیند نیز گسترش داد.

۲/۴/۲ چالش‌های گسترش به N فرآیند

گسترش الگوریتم پترسون به N فرآیند کمی پیچیده‌تر از حالت دو فرآیندی است. برخی از چالش‌ها عبارتند از:

- ساختار داده‌ها: با افزایش تعداد فرآیندها، ساختار داده‌های مورد استفاده برای هماهنگی بین فرآیندها پیچیده‌تر می‌شود.
- پیچیدگی الگوریتم: افزایش تعداد فرآیندها می‌تواند منجر به افزایش پیچیدگی الگوریتم و احتمال خطا شود.
- گسترش مفهوم نوبت: در حالت دو فرآیندی، مفهوم نوبت به سادگی قابل درک است. اما در حالت N فرآیندی، باید مکانیزمی برای تعیین نوبت هر فرآیند ایجاد شود.

۲/۴/۳ پیاده‌سازی در جاوا

برای پیاده‌سازی الگوریتم پترسون برای N فرآیند در جاوا، می‌توانیم از یک آرایه برای ذخیره اطلاعات مربوط به هر فرآیند استفاده کنیم، هر عنصر از این آرایه شامل یک پرچم و یک متغیر شمارنده است. پرچم نشان می‌دهد که آیا فرآیند می‌خواهد وارد بخش بحرانی شود یا خیر و متغیر شمارنده برای تعیین نوبت هر فرآیند استفاده می‌شود.

۲/۴/۴ توضیح کد

- N : تعداد فرآیندها
- $flag[i]$: نشان می‌دهد که آیا فرآیند i می‌خواهد وارد بخش بحرانی شود یا خیر
- $turn[i]$: نشان می‌دهد که نوبت به کدام فرآیند است
- $enterRegion$: فرآیند جاری به همه فرآیندهای دیگر اعلام می‌کند که قصد ورود به بخش بحرانی را دارد.
- فرآیند جاری منتظر می‌ماند تا همه فرآیندهای دیگر از بخش بحرانی خارج شوند و نوبت به او برسد.
- فرآیند جاری وارد بخش بحرانی می‌شود.
- $leaveRegion$: فرآیند جاری پس از خروج از بخش بحرانی، پرچم خود را پایین می‌آورد تا فرآیندهای دیگر بتوانند وارد بخش بحرانی شوند.

برای استفاده از الگوریتم پترسون، به یک تابع `main` نیاز داریم تا فرآیندها را شبیه‌سازی کند و نحوه استفاده از این الگوریتم را نشان دهد.

- در اول `numProcesses` تعداد فرآیندها را مشخص می‌کند.
- یک نمونه از کلاس `PetersonAlgorithm` ایجاد می‌کند.
- برای هر فرآیند یک رشته جداگانه ایجاد می‌کند.
- در هر رشته، یک حلقه بی‌نهایت اجرا می‌شود که فرآیند را به طور مداوم وارد و خارج از بخش بحرانی می‌کند.
- در بخش بحرانی، یک پیام چاپ می‌شود و سپس یک مکث کوتاه برای شبیه‌سازی کار در بخش بحرانی ایجاد می‌شود.

```

1 package javaapp;
2 /* Peterson Algorithm
3  * @author Shahriar
4  * @date Nov 25, 2024
5 */
6 public class JavaApp {
7     public static void main(String[] args) {
8         int numProcesses = 5;
9         PetersonAlgorithm peterson = new PetersonAlgorithm(numProcesses);
10        Thread[] threads = new Thread[numProcesses];
11        for (int i = 0; i < numProcesses; i++) {
12            int processId = i;
13            threads[i] = new Thread(() -> {
14                while (true) {
15                    peterson.enterRegion(processId);
16                    System.out.println("Process " + processId + " is in critical region.");
17                    try {
18                        Thread.sleep(1000);
19                    } catch (InterruptedException e) {
20                        e.printStackTrace();
21                    }
22                    peterson.leaveRegion(processId);
23                    System.out.println("Process " + processId + " is in not-critical region.");
24                }
25            });
26            threads[i].start();
27        }
28    }
29 }
30
31 class PetersonAlgorithm {
32     private int N;
33     private boolean[] flag;
34     private int[] turn;
35     public PetersonAlgorithm(int N) {
36         this.N = N;
37         flag = new boolean[N];
38         turn = new int[N];
39         for (int i = 0; i < N; i++) {
40             flag[i] = false;
41             turn[i] = 0;
42         }
43     }
44     public void enterRegion(int process) {
45         for (int j = 0; j < N; j++) {
46             if (j != process) {
47                 turn[j] = process;
48                 while (flag[j] && turn[j] == process);
49             }
50         }
51         flag[process] = true;
52     }
53     public void leaveRegion(int process) {
54         flag[process] = false;
55     }
56 }

```

شکل ۴- پیاده‌سازی الگوریتم پترسون در جاوا

```
1 PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin4.JavaApp'
2 Process 0 is in critical region.
3 Process 0 is in not-critical region.
4 Process 1 is in critical region.
5 Process 1 is in not-critical region.
6 Process 2 is in critical region.
7 Process 2 is in not-critical region.
8 Process 3 is in critical region.
9 Process 3 is in not-critical region.
10 Process 4 is in critical region.
11 Process 4 is in not-critical region.
12 Process 0 is in critical region.
13 Process 0 is in not-critical region.
```

شکل ۵- خروجی الگوریتم پترسون با ۵ پروسه

۲/۵ تمرین ۵ – الگوریتم نانوایی

الگوریتم نانوایی چگونه پیاده سازی می‌شود. کدها را به یک زبان برنامه نویسی دلخواه پیاده سازی و گزارش آن (توضیحات مربوط به الگوریتم نانوایی) را با LaTeX تهیه نمایید.

۲/۵/۱ الگوریتم نانوایی

الگوریتم نانوایی یک روش نرم افزاری برای حل مسئله انحصار متقابل در سیستم‌های همروند است. این الگوریتم به چندین فرآیند اجازه می‌دهد تا به صورت ایمن از یک منبع مشترک استفاده کنند بدون اینکه به داده‌های مشترک آسیب برسد. الگوریتم نانوایی به این دلیل به این نام نامگذاری شده است که مشابه صفت انتظار در یک نانوایی عمل می‌کند؛ هر فرآیند یک شماره می‌گیرد و منتظر می‌شود تا نوبت او برسد. در کل مراحل الگوریتم نانوایی به شرح زیر است:

- شماره انتخاب: هر فرآیند یک شماره انتخاب می‌کند. این شماره نشان‌دهنده تمایل فرآیند برای ورود به بخش بحرانی است.
- شماره نوبت: فرآیندها با توجه به شماره انتخاب خود به ترتیب وارد بخش بحرانی می‌شوند. فرآیندی که کوچکترین شماره انتخاب را دارد، اولویت ورود به بخش بحرانی را دارد.
- شکستن تساوی: اگر دو یا چند فرآیند دارای شماره انتخاب یکسانی باشند، از یک مکانیزم اضافی برای شکستن تساوی استفاده می‌شود.

۲/۵/۲ توضیح کد

- `number[i]`: شماره انتخاب فرآیند i
- `choosing[i]`: نشان می‌دهد که آیا فرآیند i در حال انتخاب شماره جدید است یا خیر

مراحل ورود به بخش بحرانی:

- فرآیند یک شماره انتخاب می‌کند که بزرگتر از تمام شماره‌های انتخاب شده قبلی است.
- فرآیند منتظر می‌ماند تا همه فرآیندهای دیگر شماره انتخاب خود را مشخص کنند.
- فرآیند بررسی می‌کند که آیا شماره انتخاب شده آن کمتر از سایر فرآیندها است یا اگر برابر باشد، آیا شماره شناسایی فرآیند آن کمتر است. در این صورت، فرآیند وارد بخش بحرانی می‌شود.
- مرحله خروج از بخش بحرانی: فرآیند شماره انتخاب خود را به صفر تغییر می‌دهد تا نشان دهد که از بخش بحرانی خارج شده است.

برای استفاده از الگوریتم نانوایی، به یک تابع `main` نیاز داریم تا فرآیندها را شبیه‌سازی کند و نحوه استفاده از این الگوریتم را نشان دهد.

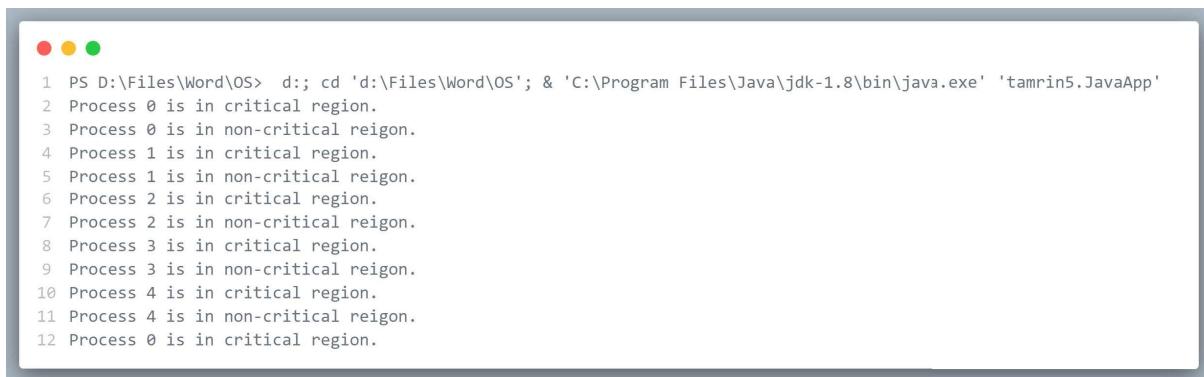
- تعداد فرآیندها را مشخص می‌کند.
- یک نمونه از کلاس `BakersAlgorithm` ایجاد می‌کند.
- برای هر فرآیند یک رشته جداوله ایجاد می‌کند.
- در هر رشته، یک حلقه بی‌نهایت اجرا می‌شود که فرآیند را به طور مداوم وارد و خارج از بخش بحرانی می‌کند.
- در بخش بحرانی، یک پیام چاپ می‌شود و سپس یک مکث کوتاه برای شبیه‌سازی کار در بخش بحرانی ایجاد می‌شود.

```

● ● ●
1 package javaapp;
2 /* Peterson Algorithm
3 * @author Shahriar
4 * @date Nov 25, 2024
5 */
6 public class JavaApp {
7     public static void main(String[] args) {
8         int numProcesses = 5;
9         BakersAlgorithm bakers = new BakersAlgorithm(numProcesses);
10        Thread[] threads = new Thread[numProcesses];
11        for (int i = 0; i < numProcesses; i++) {
12            int processId = i;
13            threads[i] = new Thread(() -> {
14                while (true) {
15                    bakers.enterRegion(processId);
16                    System.out.println("Process " + processId + " is in critical region.");
17                    try {
18                        Thread.sleep(1000);
19                    } catch (InterruptedException e) {
20                        e.printStackTrace();
21                    }
22                    bakers.leaveRegion(processId);
23                    System.out.println("Process " + processId + " is in non-critical reigon.");
24                }
25            });
26            threads[i].start();
27        }
28    }
29 }
30 class BakersAlgorithm {
31     private int N;
32     private int[] number;
33     private boolean[] choosing;
34     public BakersAlgorithm(int N) {
35         this.N = N;
36         number = new int[N];
37         choosing = new boolean[N];
38     }
39     public void enterRegion(int process) {
40         int max = 0;
41         choosing[process] = true;
42         number[process] = 1 + maxFor(number);
43         choosing[process] = false;
44         for (int j = 0; j < N; j++) {
45             while (choosing[j]);
46             while (number[j] != 0 && (number[j] < number[process] || (number[j] == number[process] && j < process)));
47         }
48     }
49     public void leaveRegion(int process) {
50         number[process] = 0;
51     }
52     private int maxFor(int[] array) {
53         int max = 0;
54         for (int i = 0; i < array.length; i++) {
55             max = Math.max(max, array[i]);
56         }
57         return max;
58     }
59 }
60

```

شکل ۶- الگوریتم نابوابی



```
PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin5.JavaApp'
1 Process 0 is in critical region.
2 Process 0 is in non-critical region.
3 Process 0 is in critical region.
4 Process 1 is in critical region.
5 Process 1 is in non-critical region.
6 Process 2 is in critical region.
7 Process 2 is in non-critical region.
8 Process 3 is in critical region.
9 Process 3 is in non-critical region.
10 Process 4 is in critical region.
11 Process 4 is in non-critical region.
12 Process 0 is in critical region.
```

شکل ۷- اجرای الگوریتم نانوایی با ۵ پروسه

شایان ذکر است گزارش بیاده‌سازی الگوریتم نانوایی بطور کامل و به زبان انگلیسی^۱ در برنامه لاتک نوشته شده و پیوست این فایل خواهد شد.

^۱ بدلیل مشکلات و عدم راه اندازی درست XePersian امکان کامپایل آن به زبان فارسی محدود نبود.

۲/۶ تمرین ۶ - مسئله تولیدکننده مصرفکننده و شام فیلسوفان

مسئله تولیدکننده مصرفکننده و مسئله غذا خوردن فیلسوفان را با استفاده از مانیتورها پیاده سازی نمایید.

۲/۶/۱ مسئله تولیدکننده و مصرفکننده با استفاده از مانیتور در جawa

مسئله تولیدکننده و مصرفکننده کلاسیک در همووندی است که در آن یک یا چند فرآیند (تولیدکننده) داده‌هایی را تولید می‌کنند که توسط یک یا چند فرآیند دیگر (مصرفکننده) مصرف می‌شوند. این مسئله برای مدل‌سازی بسیاری از سیستم‌های واقعی مانند صف‌های انتظار، بافرها و پایگاه‌های داده استفاده می‌شود.

۲/۶/۲ راه حل با استفاده از مانیتور:

مانیتورها مکانیزمی برای کنترل دسترسی به منابع مشترک در برنامه‌های همووند هستند. در جawa، می‌توانیم از مانیتورها برای حل مسئله تولیدکننده و مصرفکننده استفاده کنیم.

۲/۶/۳ توضیح کد:

- `:buffer`: آرایه‌ای برای ذخیره داده‌ها
- `:count`: تعداد عناصر موجود در بافر
- `:in`: شاخصی برای قرار دادن داده جدید در بافر
- `:out`: شاخصی برای برداشتن داده از بافر

متدها:
`:enter`

- اگر بافر پر باشد، رشته منتظر می‌ماند.
- داده جدید را به بافر اضافه می‌کند.
- متدهای `notifyAll` را فراخوانی می‌کند تا رشته‌های منتظر را بیدار کند.

متدها:
`:remove`

- اگر بافر خالی باشد، رشته منتظر می‌ماند.
- داده‌ای را از بافر برمهی دارد.
- متدهای `notifyAll` را فراخوانی می‌کند تا رشته‌های منتظر را بیدار کند.
- رشته‌های تولیدکننده و مصرفکننده: هر رشته به طور مداوم داده‌ها را تولید یا مصرف می‌کند.

```

1 package javaapp;
2 /* Producer & Consumer Algorithm
3  * @author Shahriar
4  * @date Nov 25, 2024
5 */
6 public class JavaApp {
7     public static void main(String[] args) {
8         ProducerConsumer pc = new ProducerConsumer(5);
9         Thread producer = new Thread(() -> {
10             for (int i = 0; i < 10; i++) {
11                 pc.enter(i);
12                 System.out.println("Producer: " + i);
13             }
14         });
15         Thread consumer = new Thread(() -> {
16             for (int i = 0; i < 10; i++) {
17                 int item = pc.remove();
18                 System.out.println("Consumer: " + item);
19             }
20         });
21         producer.start();
22         consumer.start();
23     }
24 }
25 class ProducerConsumer {
26     private int[] buffer;
27     private int count;
28     private int in;
29     private int out;
30     public ProducerConsumer(int size) {
31         buffer = new int[size];
32         count = 0;
33         in = 0;
34         out = 0;
35     }
36     public synchronized void enter(int item) {
37         while (count == buffer.length) {
38             try {
39                 wait();
40             } catch (InterruptedException e) {
41                 e.printStackTrace();
42             }
43         }
44         buffer[in] = item;
45         in = (in + 1) % buffer.length;
46         count++;
47         notifyAll();
48     }
49
50     public synchronized int remove() {
51         while (count == 0) {
52             try {
53                 wait();
54             } catch (InterruptedException e) {
55                 e.printStackTrace();
56             }
57         }
58         int item = buffer[out];
59         out = (out + 1) % buffer.length;
60         count--;
61         notifyAll();
62         return item;
63     }
64 }
65 
```

شکل ۱- تولید کنند مصرف کننده با استفاده از سمافور

```
● ● ●  
1 PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin6_1.JavaApp'  
2 Producer: 0  
3 Producer: 1  
4 Consumer: 0  
5 Producer: 2  
6 Producer: 3  
7 Producer: 4  
8 Producer: 5  
9 Producer: 6  
10 Consumer: 1  
11 Consumer: 2  
12 Producer: 7  
13 Producer: 8  
14 Consumer: 3  
15 Consumer: 4  
16 Producer: 9  
17 Consumer: 5  
18 Consumer: 6  
19 Consumer: 7  
20 Consumer: 8  
21 Consumer: 9
```

شکل ۹- اجرای الگوریتم تولیدکننده مصرف‌کننده با بافر ۵ و ۱۰ بروسه

۲/۶/۴ پیاده‌سازی مسئله غذا خوردن فیلسوفان با استفاده از مانیتور در جاوا

مشکل غذا خوردن فیلسوفان یک مسئله کلاسیک در همومندی است که برای مدل‌سازی منابع مشترک و مشکلات همگام‌سازی استفاده می‌شود. در این مسئله، تعدادی فیلسوف دور یک میز نشسته‌اند و هر فیلسوف به دو چنگال نیاز دارد تا غذا بخورد. چنگال‌ها منابع مشترک هستند و هر فیلسوف باید هر دو چنگال مجاور خود را بردارد تا بتواند غذا بخورد.

۲/۶/۵ توضیح کد

Philosopher کلاس

- N: تعداد فیلسوفان را مشخص می‌کند.
- State: وضعیت هر فیلسوف مبنی بر فکر کردن، گرسنگی و یا غذاخوردن را نشان می‌دهد.
- S: یک شی برای نمایش فیلسوف در نظر گرفته شده است.
- تابع سازنده: تعداد فیلسوفان را می‌گیرد، برای هر کدام یک شی می‌سازد و وضعیت اولیه هر کدام را فکر کردن مقداردهی می‌کند.
- Take_forks: اگر فیلسوف گرسنه باشد و بتواند قاشق بردارد، اجازه غذاخوردن دارد.
- Put_forks: وقتی فیلسوف کار خود را به اتمام می‌رساند، قاشق‌ها را سرجای خود می‌گذارد.
- Test: در صورت گرسنگی فیلسوف و در صورتی که اطرافیانش در حال خوردن نباشند، فیلسوف می‌تواند غذا بخورد.
- Right و Left: مقدار چپ و راست فیلسوف را بر می‌گرداند.

کلاس اصلی

- به تعداد N فیلسوف فکر می‌کنند، می‌خورند و دوباره از اول شروع می‌کنند.



```

1 PS D:\Files\Word\OS> d:; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin6_2.JavaApp'
2 Philosopher 0 is thinking...
3 Philosopher 1 is thinking...
4 Philosopher 2 is thinking...
5 Philosopher 3 is thinking...
6 Philosopher 4 is thinking...
7 Philosopher 0 is eating...
8 Philosopher 0 is thinking...
9 Philosopher 1 is eating...
10 Philosopher 1 is thinking...
11 Philosopher 2 is eating...
12 Philosopher 2 is thinking...
13 Philosopher 3 is eating...
14 Philosopher 3 is thinking...
15 Philosopher 4 is eating...
16 Philosopher 4 is thinking...
17 Philosopher 0 is eating...

```

شکل ۱۰- اجرای الگوریتم با ۵ فیلسوف

```

● ● ●
1 package tamrin6_2;
2 /* Phiosopher Dinner
3 * @author Shahrir
4 * @date Nov 25, 2024
5 */
6 public class JavaApp {
7     public static void main(String[] args) {
8         int N = 5;
9         Philosopher philosopher = new Philosopher(N);
10        Thread[] t = new Thread[N];
11
12        for (int i = 0; i < N; i++) {
13            final int philosopherId = i; // Important: Make i final for lambda
14            t[i] = new Thread(() -> {
15                while (true) {
16                    System.out.println("Philosopher " + philosopherId + " is thinking...");
17                    try {
18                        Thread.sleep(1000);
19                    } catch (InterruptedException e) {
20                        Thread.currentThread().interrupt(); // Restore interrupt status
21                        return;
22                    }
23                    philosopher.take_forks(philosopherId);
24                    System.out.println("Philosopher " + philosopherId + " is eating...");
25                    try {
26                        Thread.sleep(1000);
27                    } catch (InterruptedException e) {
28                        Thread.currentThread().interrupt(); // Restore interrupt status
29                        return;
30                    }
31                    philosopher.put_forks(philosopherId);
32                }
33            });
34            t[i].start();
35        }
36    }
37 }
38
39 class Philosopher {
40     int N;
41     int[] state;
42     Object[] S;
43     final int thinking = 0;
44     final int hungry = 1;
45     final int eating = 2;
46
47     Philosopher(int N) {
48         this.N = N;
49         state = new int[N];
50         S = new Object[N];
51         for (int i = 0; i < N; i++) {
52             state[i] = thinking;
53             S[i] = new Object();
54         }
55     }
56
57     synchronized void take_forks(int i) {
58         state[i] = hungry;
59         // Always try to acquire the left fork first
60         test(left(i));
61         if (state[i] != eating) {
62             test(right(i)); // Only try for right fork if left is acquired
63             if (state[i] != eating) {
64                 try {
65                     S[i].wait();
66                 } catch (InterruptedException e) {
67                     Thread.currentThread().interrupt();
68                     return;
69                 }
70             }
71         }
72     }
73
74     synchronized void put_forks(int i) {
75         state[i] = thinking;
76         test(left(i));
77         test(right(i));
78     }
79
80     synchronized void test(int i) {
81         if (state[i] == hungry && state[left(i)] != eating && state[right(i)] != eating && state[i] != eating) {
82             state[i] = eating;
83             S[i].notify();
84         }
85     }
86
87     int left(int i) {
88         return (i + N - 1) % N;
89     }
90
91     int right(int i) {
92         return (i + 1) % N;
93     }
94 }

```

شکل ۱۱- مسئله شام فیلسوفان با استفاده از سمافور

۲/۷ تمرين ۷ - مسئله خوانندگان و نویسندهان

مسئله خوانندگان و نویسندهان را با استفاده از مکانیزم ارسال پیام حل نمایید.

۲/۷/۱ مسئله خوانندگان و نویسندهان با استفاده از مکانیزم ارسال پیام

مسئله خوانندگان و نویسندهان یک مسئله کلاسیک در هموروندی است که در آن چندین فرآیند یا رشته به یک منبع مشترک دسترسی دارند. برخی از این فرآیندها می‌خواهند داده‌ها را بخوانند (خوانندگان) و برخی دیگر می‌خواهند داده‌ها را بنویسند (نویسنده). هدف اصلی در این مسئله این است که دسترسی به منبع مشترک به گونه‌ای مدیریت شود که از بروز خطاهای ناشی از دسترسی همزمان چندین فرآیند جلوگیری شود.

در این روش، ما از یک صفت پیام برای هماهنگی بین خوانندگان و نویسندهان استفاده می‌کنیم. این صفت به عنوان یک واسطه عمل می‌کند و درخواست‌های خواندن و نوشتمن را مدیریت می‌کند.

۲/۷/۲ الگوریتم:

- صفت پیام: یک صفت پیام ایجاد می‌شود که درخواست‌های خواندن و نوشتمن در آن قرار می‌گیرند.
- فرآیند اصلی: یک فرآیند اصلی مسئول مدیریت صفت پیام است. این فرآیند درخواست‌ها را از صفت خارج کرده و بر اساس اولویت آن‌ها، به درخواست‌ها پاسخ می‌دهد.

۲/۷/۳ خواننده:

- درخواست خواندن را در صفت قرار می‌دهد.
- منتظر پاسخ از فرآیند اصلی می‌ماند.
- پس از دریافت پاسخ، به منبع مشترک دسترسی پیدا کرده و داده‌ها را می‌خواند.

۲/۷/۴ نویسنده:

- درخواست نوشتمن را در صفت قرار می‌دهد.
- منتظر می‌شود تا هیچ خواننده یا نویسنده دیگری به منبع دسترسی نداشته باشد.
- پس از دریافت پاسخ، به منبع مشترک دسترسی پیدا کرده و داده‌ها را می‌نویسد.



```

1 PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin7.JavaApp'
2 Writer wrote: Message 1
3 Reader 0 read: Message 1
4 Reading...
5 Writer wrote: Message 2
6 Reader 1 read: Message 2
7 Reading...
8 Writer wrote: Message 3
9 Reader 2 read: Message 3
10 Reading...
11 Writer wrote: Message 4
12 Reader 0 read: Message 4
13 Reading...
14 Reader 1 read: null
15 Reader 2 read: null

```

شکل ۱۲ - خروجی الگوریتم خواننده نویسنده

```

1 package tamrin7;
2 /* Writer & Reader with MSG
3 * @author Shahrivar
4 * @date Nov 25, 2024
5 */
6 import java.util.concurrent.BlockingQueue;
7 import java.util.concurrent.LinkedBlockingQueue;
8 import java.util.concurrent.Semaphore;
9 public class JavaApp {
10     private static final int BUFFER_SIZE = 5;
11     private static final BlockingQueue<String> buffer = new LinkedBlockingQueue<>(BUFFER_SIZE);
12     private static final Semaphore mutex = new Semaphore(1);
13     private static final Semaphore writeSemaphore = new Semaphore(1);
14     private static final Semaphore readSemaphore = new Semaphore(0);
15     private static int readCount = 0;
16
17     public static void main(String[] args) {
18         Thread writerThread = new Thread(() -> {
19             for (int i = 1; i < 5; i++) {
20                 String message = "Message " + i;
21                 writer(message);
22             }
23         });
24
25         Thread[] readerThreads = new Thread[3];
26         for (int i = 0; i < readerThreads.length; i++) {
27             final int readerId = i;
28             readerThreads[i] = new Thread(() -> {
29                 for (int j = 0; j < 5; j++) {
30                     String message = reader(readerId);
31                     if (message != null) {
32                         System.out.println("Reader " + readerId + " read: " + message);
33                     }
34                 }
35             });
36         }
37
38         writerThread.start();
39         for (Thread readerThread : readerThreads) {
40             readerThread.start();
41         }
42     }
43
44     public static void writer(String message) {
45         try {
46             writeSemaphore.acquire();
47             mutex.acquire();
48             buffer.put(message);
49             System.out.println("Writer wrote: " + message);
50             mutex.release();
51             readSemaphore.release(1);
52         } catch (InterruptedException e) {
53             Thread.currentThread().interrupt();
54         } finally {
55             writeSemaphore.release();
56         }
57     }
58
59     public static String reader(int readerId) {
60         try {
61             readSemaphore.acquire();
62             mutex.acquire();
63             String message = buffer.poll();
64             if (message != null) {
65                 readCount++;
66             }
67             if (readCount == 1) {
68                 writeSemaphore.acquire();
69             }
70             mutex.release();
71
72             if (message != null) {
73                 return message;
74             } else {
75                 return null;
76             }
77         } catch (InterruptedException e) {
78             Thread.currentThread().interrupt();
79             return null;
80         } finally {
81             try {
82                 mutex.acquire();
83                 readCount--;
84                 if (readCount == 0) {
85                     writeSemaphore.release();
86                 }
87                 mutex.release();
88             } catch (InterruptedException e) {
89                 Thread.currentThread().interrupt();
90             }
91         }
92     }
93
94     public static void write_section() {
95         System.out.println("Writing...");
96     }
97
98     public static void Read_section() {
99         System.out.println("Reading...");
100    }
101 }
102

```

شکل ۱۳- الگوریتم خواننده نویسنده به روش ارسال پیام

۲/۸ تمرین ۸ - مقاله

از آنجایی که حجم و سطح مقاله بالا می‌باشد و خارج از حوصله ترجمه و تفهیم می‌باشد، از دستیارها و هوش مصنوعی فقط جهت ترجمه آن استفاده شده تا خلاصه و چکیده در زیر ارائه شود:

هدف اصلی این متن، بررسی تاریخچه و تکامل الگوریتم‌هایی است که برای حل این مشکل ارائه شده‌اند. از زمان معرفی اولیه این مسئله در دهه ۱۹۶۰، الگوریتم‌های مختلفی با ویژگی‌ها و پیچیدگی‌های مختلف برای حل آن ارائه شده است.

مهم‌ترین نکات که در این متن به آن‌ها پرداخته شده است:

- اهمیت مسئله: انحصار متقابل در بسیاری از سیستم‌های نرمافزاری از جمله سیستم‌عامل‌ها، پایگاه داده‌ها و شبکه‌ها بسیار مهم است.
- تاریخچه: متن به طور خلاصه تاریخچه‌ای از تکامل الگوریتم‌های انحصار متقابل را ارائه می‌دهد و نشان می‌دهد که چگونه با گذشت زمان، الگوریتم‌ها پیچیده‌تر و کارآمدتر شده‌اند.
- اثبات صحت: تأکید بر اهمیت اثبات صحت الگوریتم‌های انحصار متقابل شده است، زیرا خطای در این الگوریتم‌ها می‌تواند منجر به مشکلات جدی شود.

متن اصلی به مدل فرآیندها و نحوه ارتباط آن‌ها در الگوریتم‌های انحصار متقابل می‌پردازد. این مدل شامل فرآیندهای غیرهمزمان است که از طریق ثبات‌های مشترک با هم ارتباط برقرار می‌کنند.

- ثبات‌های خواندنی/نوشتنی: این ثبات‌ها می‌توانند توسط چندین فرآیند خوانده و نوشته شوند (MWMR) یا فقط توسط یک فرآیند نوشته و توسط همه فرآیندها خوانده شوند (SWMR).
- ثبات‌های اتمی: این ثبات‌ها عملیات خواندن و نوشتمن را به صورت اتمی انجام می‌دهند، یعنی عملیات‌ها به صورت کامل و غیرقابل تقسیم اجرا می‌شوند.
- ثبات‌های غیر اتمی: این ثبات‌ها ممکن است در برخی شرایط رفتار غیرقابل پیش‌بینی داشته باشند.
- عملیات قوی‌تر: برخی پردازنده‌ها عملیات قوی‌تری مانند compare&swap و fetch&increment، swap را ارائه می‌دهند که می‌توانند برای حل مسائل همزمانی پیچیده‌تر استفاده شوند.

۲/۸/۱ انحصار متقابل (Mutex) و چالش‌های مرتبط با آن

این مسئله به این معنی است که تنها یک فرآیند در هر لحظه می‌تواند به یک منبع مشترک دسترسی داشته باشد. چالش‌های اصلی در حل این مسئله عبارتند از:

- انحصار متقابل: اطمینان حاصل شود که تنها یک فرآیند در هر لحظه به بخش بحرانی دسترسی داشته باشد.
- عدم قحطی: هیچ فرآیندی نباید به طور نامحدود منتظر بماند.
- عدم بن‌بست: باید از ایجاد شرایطی که در آن هیچ فرآیندی نتواند پیشرفت کند، جلوگیری شود.

الگوریتم‌های انحصار متقابل برای حل این مشکل طراحی شده‌اند. این الگوریتم‌ها معمولاً از عملیات اتمی مانند خواندن، نوشتمن، افزایش مقدار، تعویض مقدار و مقایسه و تعویض استفاده می‌کنند.

۲/۸/۲ مفهوم الگوریتم‌های متقابن

این الگوریتم‌ها باید به گونه‌ای طراحی شوند که فرآیندها از نظر الگوریتم یکسان باشند و تنها با استفاده از مقایسه شناسه‌های خود توانند تصمیم‌گیری کنند.

۲/۸/۳ الگوریتم انحصار متقابل دیجکسترا

این الگوریتم یکی از اولین الگوریتم‌های مطرح شده برای حل مسئله انحصار متقابل است. مفاهیم کلیدی این بخش عبارتند از:

- ثبات‌های مشترک: الگوریتم از ثبات‌های خواندنی انشتنی مشترک برای هماهنگی بین فرآیندها استفاده می‌کند.
- پرچم‌های رقابت: هر فرآیند از یک پرچم برای نشان دادن اینکه قصد ورود به بخش بحرانی را دارد استفاده می‌کند.
- ثبات NEXT: این ثبات برای تعیین فرآیند بعدی که باید وارد بخش بحرانی شود استفاده می‌شود.

الگوریتم دیجکسترا به این صورت عمل می‌کند:

- علامت‌گذاری رقابت: فرآیند پرچم خود را بالا می‌برد تا نشان دهد که می‌خواهد وارد بخش بحرانی شود.
- رقابت برای نوبت: فرآیند سعی می‌کند خود را به عنوان فرآیند بعدی تعیین کند.
- ورود به بخش بحرانی: اگر فرآیند موفق شود به عنوان فرآیند بعدی تعیین شود، وارد بخش بحرانی می‌شود.
- خروج از بخش بحرانی: پس از اتمام کار در بخش بحرانی، فرآیند پرچم خود را پایین می‌آورد.
- اثبات صحت: متن به بررسی اثبات صحت الگوریتم دیجکسترا می‌پردازد. این اثبات نشان می‌دهد که تنها یک فرآیند در هر لحظه می‌تواند در بخش بحرانی باشد.

۲/۸/۴ چالش‌های پیاده‌سازی الگوریتم‌های انحصار متقابل بر روی ثبات‌های غیر اتمی و الگوریتم لامپورت

چالش‌های استفاده از ثبات‌های غیر اتمی:

- عدم اطمینان در خواندن: اگر خواندن و نوشتمن به صورت همزمان روی یک ثبات غیر اتمی انجام شود، نتیجه خواندن ممکن است غیرقابل پیش‌بینی باشد.
- پیجیدگی پیاده‌سازی: پیاده‌سازی الگوریتم‌های انحصار متقابل بر روی ثبات‌های غیر اتمی نیازمند دقت و توجه به جزئیات است.
- الگوریتم لامپورت:

 - مفهوم تیکت: هر فرآیند یک تیکت دریافت می‌کند و منتظر می‌ماند تا نوبت آن فرا برسد.
 - حل مسئله تیکت‌های تکراری: برای جلوگیری از تکراری شدن تیکت‌ها، از شناسه فرآیندها برای ایجاد یک ترتیب کامل بین تیکت‌ها استفاده می‌شود.
 - پیاده‌سازی: الگوریتم لامپورت با استفاده از ثبات‌های غیر اتمی پیاده‌سازی شده است و از مفهوم تیکت برای حل مسئله انحصار متقابل استفاده می‌کند.

الگوریتم لامپورت برای حل مسئله انحصار متقابل بر روی ثبات‌های غیر اتمی می‌پردازد. این الگوریتم یکی از الگوریتم‌های مهم و تأثیرگذار در این حوزه است.

- ثبات‌های ایمن: این نوع ثبات‌ها ضعیفتر از ثبات‌های اتمی هستند و ممکن است در برخی شرایط رفتار غیرقابل پیش‌بینی داشته باشند.
- شماره تیکت: هر فرآیند یک شماره تیکت دریافت می‌کند که برای تعیین ترتیب ورود به بخش بحرانی استفاده می‌شود.
- مقایسه تیکت‌ها: فرآیندها با مقایسه شماره تیکت‌های خود، مشخص می‌کنند که کدام فرآیند باید به بخش بحرانی وارد شود.

۲/۸/۵ الگوریتم لامپورت:

- دریافت تیکت: هر فرآیند یک شماره تیکت دریافت می‌کند.
- انتظار: فرآیند منتظر می‌ماند تا نوبت آن فرا برسد. این کار با مقایسه شماره تیکت خود با شماره تیکت سایر فرآیندها انجام می‌شود.
- ورود به بخش بحرانی: اگر فرآیند دارای کوچک‌ترین شماره تیکت باشد، وارد بخش بحرانی می‌شود.

- خروج از بخش بحرانی: پس از اتمام کار در بخش بحرانی، فرآیند شماره تیکت خود را صفر می‌کند.

۲/۸/۶ اثبات صحت:

- عدم قحطی: الگوریتم لامپورت خاصیت عدم قحطی را تضمین می‌کند، یعنی هیچ فرآیندی به طور نامحدودمنتظر نمی‌ماند.
- انحصار متقابل: الگوریتم تضمین می‌کند که در هر لحظه تنها یک فرآیند در بخش بحرانی باشد.

الگوریتم نانوایی لامپورت یک الگوریتم کلاسیک در حوزه محاسبات موازی و توزیع شده است که برای حل مسئله انحصار متقابل به کار می‌رود. این الگوریتم به دلیل سادگی و کارایی، تأثیر قابل توجهی بر درک ما از سیستم‌های پیام‌گذاری ناهمزمان داشته است. مهم‌ترین نوآوری‌های این الگوریتم عبارتند از:

- استفاده از ثبات‌های SWMR: این الگوریتم از ثبات‌هایی استفاده می‌کند که تنها یک فرآیند می‌تواند در آن‌ها بنویسد اما چندین فرآیند می‌تواند از آن‌ها بخواهد. این نوع ثبات‌ها پیاده‌سازی ساده‌ای دارند.
- معرفی مفهوم مهر زمان: این الگوریتم مفهوم مهر زمان را معرفی می‌کند که برای ردیابی ترتیب وقایع در یک سیستم ناهمزمان به کار می‌رود. مهر زمان به ما کمک می‌کند تا علیت را در سیستم‌های توزیع شده حفظ کنیم.
- پایه‌های محاسبات بدون انتظار: تکنیک‌هایی به کار رفته در این الگوریتم، به ویژه استفاده از ثبات‌های ایمن و خواندن همزمان هنگام نوشتمن، پایه و اساس محاسبات بدون انتظار را فراهم کرده‌اند. در محاسبات بدون انتظار، فرآیندها می‌توانند بدون اینکه منتظر فرآیند دیگری بمانند، پیش‌رفت کنند.

۲/۸/۷ الگوریتم انحصار متقابل پتروسون

این الگوریتم یک الگوریتم ساده و کارآمد برای حل مسئله انحصار متقابل در دو فرآیند است.

- ثبت‌های مشترک: الگوریتم از ثبات‌های خواندنی/نوشتمنی مشترک برای هماهنگی بین فرآیندها استفاده می‌کند.
- پرچم‌های رقابت: هر فرآیند از یک پرچم برای نشان دادن اینکه قصد ورود به بخش بحرانی را دارد استفاده می‌کند.
- ثبتات AFTERYOU: این ثبات برای تعیین فرآیند بعدی که باید وارد بخش بحرانی شود استفاده می‌شود.
- علامت‌گذاری رقابت: فرآیند پرچم خود را بالا می‌برد.
- اعلام آمادگی: فرآیند مشخص می‌کند که آماده ورود به بخش بحرانی است.
- انتظار: فرآیند منتظر می‌شود تا فرآیند دیگر از بخش بحرانی خارج شود یا اعلام کند که قصد ورود ندارد.
- ورود به بخش بحرانی: اگر شرایط مناسب باشد، فرآیند وارد بخش بحرانی می‌شود.
- خروج از بخش بحرانی: پس از اتمام کار در بخش بحرانی، فرآیند پرچم خود را پایین می‌آورد.

توسعه به n فرآیند:

- مفهوم پله‌ها: الگوریتم برای n فرآیند به صورت یک نرده‌بان در نظر گرفته می‌شود. هر فرآیند باید از پله‌های این نرده‌بان بالا برود تا به بخش بحرانی برسد.
- رقابت در هر پله: در هر پله، فرآیندها با استفاده از مکانیزم مشابه الگوریتم دو فرآیندی رقابت می‌کنند.
- تضمين انصاف: الگوریتم به گونه‌ای طراحی شده است که از قحطی جلوگیری کند و هر فرآیند در نهایت بتواند به بخش بحرانی دسترسی پیدا کند.

در بررسی چگونگی تعمیم الگوریتم پتروسون برای حل مسئله چند انحصار متقابل، در حالی که در مسئله انحصار متقابل معمولی (mutex)، تنها یک فرآیند می‌تواند به یک بخش بحرانی دسترسی داشته باشد، در مسئله چند انحصار متقابل، حداقل k فرآیند می‌توانند همزمان به این بخش دسترسی پیدا کنند.

- تعمیم ساده: با یک تغییر کوچک در الگوریتم پرسون، می‌توان آن را به گونه‌ای تعمیم داد که برای مسئله ک-انحصار متقابل نیز قابل استفاده باشد. این تغییر شامل کاهش تعداد سطوح در ساختار نرdbanی الگوریتم است.
- بهبود کارایی: الگوریتم‌های اولیه برای حل مسئله انحصار متقابل، پیچیدگی زمانی خطی داشتند. اما با استفاده از تکنیک‌های مبتنی بر مسابقات (tournament)، می‌توان این پیچیدگی را به لگاریتمی کاهش داد.
- ساختار درختی مسابقات: در این روش، فرآیندها در یک ساختار درختی با هم رقابت می‌کنند تا به بخش بحرانی دسترسی پیدا کنند. این ساختار به طور قابل توجهی تعداد دسترسی‌ها به حافظه مشترک را کاهش می‌دهد.

با استفاده از تغییرات ساده و هوشمندانه در الگوریتم‌های موجود، می‌توان کارایی آن‌ها را به طور قابل توجهی بهبود بخشد. همچنین، مفهوم مسابقات و ساختارهای درختی در طراحی الگوریتم‌های همگام‌سازی نقش بسیار مهمی دارد.

- انحصار متقابل: اجازه دسترسی همزمان حداقل k فرآیند به یک بخش بحرانی
- ساختار نرdbanی: ساختاری برای مدل‌سازی رقابت بین فرآیندها
- مسابقات: روشی برای کاهش تعداد دسترسی‌ها به حافظه مشترک
- درخت مسابقات: ساختاری درختی برای پیاده‌سازی مسابقات

۲/۸/۸ الگوریتم انحصار متقابل فیشر

- فرض زمان واقعی: در این الگوریتم، فرض می‌شود که بین دو دسترسی متوالی یک فرآیند به حافظه مشترک، حداقل زمان مشخصی () می‌گذرد.
- عملیات تاخیر: فرآیندها می‌توانند با استفاده از عملیات (d) delay برای مدت زمان مشخصی متوقف شوند.
- انتظار برای دسترسی: فرآیندمنتظر می‌شود تا ثبات مشترک X خالی شود.
- نوشتن شناسه: فرآیند شناسه خود را در ثبات X می‌نویسد.
- تاخیر: فرآیند برای مدت زمان منتظر می‌ماند.
- بررسی تداخل: فرآیند بررسی می‌کند که آیا شناسه آن هنوز در ثبات X وجود دارد. اگر بله، وارد بخش بحرانی می‌شود. در غیر این صورت، به مرحله اول بازمی‌گردد.

۲/۸/۹ مفهوم انعطاف‌پذیری در الگوریتم‌های انحصار متقابل

هدف اصلی، کاهش هزینه عملیات () acquire و () release در شرایطی است که رقابت کمی بین فرآیندها وجود دارد.

- هزینه عملیات: هزینه یک عملیات به تعداد دسترسی‌های آن به حافظه مشترک بستگی دارد.
- شرایط مطلوب: زمانی که یک فرآیند می‌خواهد وارد بخش بحرانی شود و هیچ فرآیند دیگری در حال رقابت نیست.
- الگوریتم‌های قبلی: الگوریتم‌های قبلی مانند دیجکسترا و لامپورت در شرایط رقابت، هزینه بالایی داشتند.
- مفهوم Splitter: یک شیء که به فرآیندها اجازه می‌دهد تا برنده، بازنده دیررس یا بازنده همزمان شوند.

۲/۸/۱۰ الگوریتم انعطاف‌پذیر:

- کاهش هزینه در شرایط مطلوب: در شرایط مطلوب، فرآیند می‌تواند با تعداد بسیار کمی دسترسی به حافظه مشترک، وارد بخش بحرانی شود.
- استفاده از Splitter: از شیء Splitter برای تعیین برنده و بازنده‌های رقابت استفاده می‌شود.
- مکانیزم تصمیم‌گیری: فرآیندها با استفاده از عملیات خواندن و نوشتن روی ثبات‌های مشترک، تصمیم می‌گیرند که آیا می‌توانند وارد بخش بحرانی شوند یا خیر.

۲/۸/۱۱ اسپین محلی در الگوریتم‌های انحصار متقابل

این تکنیک برای بهبود کارایی الگوریتم‌ها در سیستم‌های حافظه اشتراکی با حافظه نهان استفاده می‌شود.

- حافظه نهان: یک حافظه سریع که در هر پردازنده وجود دارد و کپی‌هایی از داده‌های پرکاربرد را ذخیره می‌کند.
- اسپین محلی: به جای اینکه فرآیند به طور مداوم به حافظه اصلی دسترسی کند، می‌تواند به حافظه نهان خود مراجعه کند که بسیار سریع‌تر است.
- کاهش ترافیک شبکه: با استفاده از اسپین محلی، ترافیک شبکه بین پردازنده‌ها کاهش می‌یابد.

۲/۸/۱۲ الگوریتم‌های مبتنی بر اسپین محلی:

الگوریتم‌های انحصار متقابل مبتنی بر اسپین محلی از مزایای حافظه نهان استفاده می‌کنند تا کارایی خود را بهبود بخشنند. این الگوریتم‌ها به جای اینکه به طور مداوم به حافظه اصلی دسترسی کنند، بیشتر اوقات به حافظه نهان خود مراجعه می‌کنند.

۲/۸/۱۳ فرآیندهای ناشناس و حافظه ناشناس در سیستم‌های همزمان

- فرآیندهای ناشناس: فرآیندهایی که هیچ شناسه‌ای ندارند و از نظر الگوریتم یکسان هستند.
- حافظه ناشناس: حافظه‌ای که هر فرآیند دیدگاه متفاوتی از آن دارد و آدرس‌ها برای هر فرآیند متفاوت تفسیر می‌شوند.
- انحصار متقابل در سیستم‌های ناشناس: چالش حل مسئله انحصار متقابل در سیستم‌هایی که فرآیندها و حافظه ناشناس هستند.
- شرایط لازم و کافی: برای حل مسئله انحصار متقابل در سیستم‌های ناشناس، نیاز به شرایط خاصی در مورد تعداد ثبات‌های مشترک است.
- محدودیت‌های پیاده‌سازی: به دلیل ناشناس بودن فرآیندها و حافظه، پیاده‌سازی الگوریتم‌های انحصار متقابل در این سیستم‌ها چالش‌برانگیز است.

۲/۸/۱۴ نتیجه

- انحصار متقابل در سیستم‌های با خرابی فرآیند: این بخش به چالش‌های حل مسئله انحصار متقابل در سیستم‌هایی که فرآیندها ممکن است چار خرابی شوند، پرداخته است. برخی از راهکارهای پیشنهادی شامل استفاده از حافظه غیر فرار و مکانیزم‌های بازیابی پس از خرابی است.
- انحصار متقابل گروهی: این مفهوم تعمیمی از انحصار متقابل است که به چندین گروه از فرآیندها اجازه می‌دهد به طور همزمان به یک منبع مشترک دسترسی داشته باشند.
- انحصار متقابل در سیستم‌های پیام‌گذرنگی: این بخش به بررسی الگوریتم‌های انحصار متقابل در سیستم‌های پیام‌گذرنگی ناهمzman پرداخته است.

۲/۹ تمرین اضافه - سیستم عامل‌های بلادرنگ

۲/۹/۱ سیستم عامل‌های بلادرنگ سخت و نرم

سیستم عامل‌های بلادرنگ (RTOS) یا Real-time Operating Systems به دو دسته اصلی تقسیم می‌شوند: سخت و نرم. هر دو نوع این سیستم عامل‌های برای اجرای برنامه‌هایی طراحی شده اند که نیاز به پاسخگویی سریع و دقیق در زمان مشخصی دارند، اما تفاوت‌های کلیدی بین آنها وجود دارد.

۲/۹/۲ سیستم عامل‌های بلادرنگ سخت (Hard Real-time Systems)

این نوع سیستم عامل‌های برای کاربردهایی طراحی شده اند که در آن، تأخیر در پاسخگویی به یک رویداد می‌تواند منجر به نتایج فاجعه آمیز شود. به عنوان مثال، سیستم‌های کنترل پرواز هوایی، سیستم‌های کنترل هسته‌ای و سیستم‌های پزشکی از این دسته هستند. در این سیستم‌ها، هر وظیفه باید در زمان مشخص شده خود اجرا شود و هرگونه تأخیر غیرقابل قبول است. ویژگی‌های اصلی سیستم‌های بلادرنگ سخت عبارتند از:

- زمانبندی دقیق: وظایف با اولویت بالا و زمانبندی دقیق اجرا می‌شوند.
- عدم تحمل خطأ: هرگونه خطأ یا تأخیر می‌تواند به سیستم آسیب جدی وارد کند.
- پیش‌بینی پذیری: رفتار سیستم باید قابل پیش‌بینی باشد و پاسخ به رویدادها در زمان مشخصی رخ دهد.

۲/۹/۳ سیستم عامل‌های بلادرنگ نرم (Soft Real-time Systems)

سیستم عامل‌های نرم برای کاربردهایی طراحی شده اند که در آن، تأخیر در پاسخگویی به یک رویداد می‌تواند منجر به کاهش عملکرد یا کیفیت سیستم شود، اما فاجعه آمیز نیست. به عنوان مثال، سیستم‌های چندرسانه‌ای، بازی‌های کامپیوتری و سیستم‌های کنترل صنعتی از این دسته هستند. در این سیستم‌ها، تلاش می‌شود تا وظایف در زمان مشخص شده خود اجرا شوند، اما تأخیرهای جزئی قابل تحمل هستند. ویژگی‌های اصلی سیستم‌های بلادرنگ نرم عبارتند از:

- زمانبندی تقریبی: وظایف با اولویت بالا اجرا می‌شوند، اما ممکن است تأخیرهای جزئی رخ دهد.
- تحمل خطأ: سیستم می‌تواند برخی از خطاهای تأخیرها را تحمل کند.
- انعطاف‌پذیری: این سیستم‌ها معمولاً انعطاف‌پذیرتر از سیستم‌های بلادرنگ سخت هستند.

FreeRTOS ۲/۹/۴

یک سیستم عامل‌های بلادرنگ FreeRTOS یا Real-Time Operating System (RTOS) متن باز و رایگان است که به طور گسترده در سیستم‌های نهفته (Embedded Systems) استفاده می‌شود. این سیستم عامل به گونه‌ای طراحی شده است که بتواند چندین وظیفه (Task) را به طور همزمان مدیریت کند و به هر یک از آن‌ها در بازه‌های زمانی مشخصی زمان پردازنه اختصاص دهد. از جمله مزایای این سیستم عامل عبارت اند از:

- رایگان و متن باز: FreeRTOS تحت پروانه MIT منتشر شده است که استفاده و اصلاح آن را برای همه آزاد می‌گذارد.
- سبک و کارآمد: FreeRTOS برای سیستم‌های با منابع محدود طراحی شده است و به حداقل حافظه و پردازشگر نیاز دارد.
- انعطاف‌پذیر: امکان سفارشی‌سازی و توسعه آن برای نیازهای مختلف وجود دارد.
- پشتیبانی گسترده: جامعه کاربری بزرگ و مستندات کاملی برای FreeRTOS وجود دارد.
- پشتیبانی از معماری‌های مختلف: FreeRTOS از طیف گسترده‌ای از معماری‌های پردازنده پشتیبانی می‌کند.

ویژگی‌های کلیدی FreeRTOS ۲/۹/۵

مدیریت وظایف: این امکان را می‌دهد تا وظایف مختلفی را تعریف شوند و هر یک را با اولویت و دوره زمانی مشخصی اجرا کرد.

- همگام‌سازی وظایف: برای ارتباط بین وظایف و جلوگیری از شرایط بحرانی، FreeRTOS مکانیزم‌هایی مانند متغیرهای دوگانه، سَمَفُورها و صفاتی پیام را فراهم می‌کند.
- تایمِرها: امکان ایجاد تایمِرها نرمافزاری را فراهم می‌کند که برای ایجاد وقفه‌های دوره‌ای استفاده می‌شوند.
- مدیریت حافظه: امکان مدیریت حافظه پویا را فراهم می‌کند و این اجازه را می‌دهد تا حافظه را به صورت دینامیک تخصیص و آزاد کرد.
- مدیریت وقفه‌ها: به شما امکان می‌دهد تا وقفه‌های سختافزاری را مدیریت کرده و وظایف خاصی را در پاسخ به وقفه‌ها اجرا کرد.

۲/۹/۶ کاربردهای FreeRTOS

FreeRTOS در طیف وسیعی از کاربردها از جمله موارد زیر استفاده می‌شود:

- سیستم‌های کنترل صنعتی: کنترل موتورها، سنسورها و عملگرها
- روباتیک: کنترل حرکت، حسگرها و عملگرها ربات‌ها
- اینترنت اشیا (IoT): ارتباط بین دستگاه‌های مختلف و جمع‌آوری داده‌ها
- سیستم‌های پزشکی: دستگاه‌های پزشکی قابل حمل و تجهیزات آزمایشگاهی
- خودرو: سیستم‌های کنترل خودرو و سیستم‌های اطلاعات سرگرمی

۲/۹/۷ مثالی از FreeRTOS

برای درک بهتر و اجرای FreeRTOS بروی امبد سیستم‌ها یک پروژه عملی را بررسی می‌کنیم. فرضًا یک میکروکنترلر داریم که برای ارتباط از طریق پورت سریال با دستگاه‌های دیگر طراحی شده است.

۲/۹/۸ مراحل پیاده‌سازی:

- انتخاب یک میکروکنترلر جهت پیاده‌سازی (فرضاً STM32F429)
- نصب و شروع به کار با محیط کامپایلر و توسعه (از محیط STM32CubeIDE با کتابخانه HAL استفاده می‌کنیم)
- پیکربندی و تنظیمات اولیه پروژه
- وظیفه دریافت داده از پورت سریال: این وظیفه به صورت مداوم داده‌ها را از پورت سریال می‌خواند و آن‌ها را در یک صف یا متغیر جهانی ذخیره می‌کند.
- وظیفه پردازش داده: این وظیفه داده‌های دریافتی را پردازش می‌کند (مثلاً محاسبات، تصمیم‌گیری) و نتایج را در یک متغیر یا صف دیگر ذخیره می‌کند.
- وظیفه ارسال داده از پورت سریال: این وظیفه داده‌های پردازش شده را از صف خوانده و آن‌ها را به پورت سریال ارسال می‌کند.

۲/۹/۹ توضیح کد

- صف: از یک صف برای تبادل داده بین وظایف استفاده می‌شود.
- وظایف: هر وظیفه کار مشخصی را انجام می‌دهد: دریافت داده، پردازش داده و ارسال داده.
- همگام‌سازی: وظایف با استفاده از صف با هم همگام‌سازی می‌شوند.
- پردازش داده: در این مثال، یک پردازش ساده (ضرب در ۲) انجام می‌شود. شما می‌توانید این بخش را با پردازش‌های پیچیده‌تر جایگزین کنید.

۲/۹/۱۰ مزایای این رویکرد

- مدولاریته: هر وظیفه یک کار مشخص را انجام می‌دهد که باعث می‌شود کد قابل فهم‌تر و نگهداری آسان‌تر باشد.
- انعطاف‌پذیری: می‌توان به راحتی وظایف جدیدی را اضافه کرد یا وظایف موجود را تغییر داد.

- کارایی: با تقسیم کار بین چندین وظیفه، می‌توان از منابع پردازندۀ به طور موثر استفاده کرد.

۲/۹/۱۱ نکات مهم

- انتخاب اولویت: اولویت وظایف بر اساس اهمیت آن‌ها تعیین می‌شود.
- اندازه صفت: اندازه صفت باید به اندازه کافی بزرگ باشد تا داده‌های از دست نرود.
- مدیریت خط: باید مکانیزم‌هایی برای مدیریت خطاهای احتمالی مانند سربزی شدن صفت یا خطاهای سخت‌افزاری در نظر گرفته شود.

```

● ● ●
1 /* STM32F4xx Serial FreeRTOS
2 * @author Shahriar
3 * @date Nov 25, 2024
4 */
5 #include "freertos/FreeRTOS.h"
6 #include "freertos/task.h"
7 #include "stm32f4xx_hal.h"
8
9 #define USARTX USART2
10 #define USARTX_IRQn USART2_IRQn
11
12 uint8_t rx_data;
13 uint8_t tx_data;
14
15 QueueHandle_t xQueue;
16
17 void vUARTTask(void *pvParameters) {
18     while (1) {
19         HAL_UART_Receive(&huart2, &rx_data, 1, HAL_MAX_DELAY);
20         xQueueSend(xQueue, &rx_data, portMAX_DELAY);
21     }
22 }
23
24 void vProcessTask(void *pvParameters) {
25     while (1) {
26         xQueueReceive (xQueue, &rx_data, portMAX_DELAY);
27         tx_data = rx_data * 2;
28     }
29 }
30
31 void vUARTTransmitTask(void *pvParameters) {
32     while (1) {
33         xQueueReceive (xQueue, &tx_data, portMAX_DELAY);
34         HAL_UART_Transmit(&huart2, &tx_data, 1, HAL_MAX_DELAY);
35     }
36 }
37
38 int main(void) {
39     xQueue = xQueueCreate(10, sizeof(uint8_t));
40     xTaskCreate(vUARTTask, "UARTTask", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
41     xTaskCreate(vProcessTask, "ProcessTask", configMINIMAL_STACK_SIZE, NULL, 2, NULL);
42     xTaskCreate (vUARTTransmitTask, "UARTTransmitTask", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
43     vTaskStartScheduler();
44     while (1);
45 }

```

شکل ۱۴ - پیاده‌سازی مثال ساده‌ای FreeRTOS در دریافت و ارسال اطلاعات از طریق پورت سریال

اجرای این کد بصورت واقعی بر روی سخت‌افزار پیاده می‌شود، ازین رو می‌توان آن را بروی بردھای دیسکاوری^۲ یا سری میکروکنترلرهای STM32 اجرا و نتیجه را از طریق پورت سریال مشاهده نمود.

^۲ بردھای آماده از شرکت ST برای توسعه و آموزش جهت پیاده سازی و دسترسی ساده تر می‌باشد. اطلاعات بیشتر (<https://www.st.com/>)

۲/۱۰ تمرین اضافه – پیدا کردن حفره‌های متوالی

پیدا کردن تعداد صفرهای متوالی (حفره) در حافظه

۲/۱۰/۱ هدف این تابع

هدف این کد محاسبه تعداد صفرهای متوالی در یک آرایه از بایت‌ها (اعداد ۸ بیتی بدون علامت) است. این کار با بررسی تک تک بیت‌های هر بایت و شمارش تعداد صفرهای پشت سر هم انجام می‌شود. این نوع شمارش در سیستم‌های عامل و مدیریت حافظه کاربرد دارد، به خصوص زمانی که می‌خواهیم وضعیت بلاک‌های حافظه را نمایش دهیم. فرض کنید حافظه به بلاک‌هایی تقسیم شده و هر بیت نشان‌دهنده وضعیت یک بلاک آزاد و ۱ برای بلاک اشغال شده. با استفاده از این کد می‌توانیم تعداد بلاک‌های آزاد متوالی (حفره‌های حافظه) را پیدا کنیم. این اطلاعات برای الگوریتم‌های تخصیص حافظه (مانند first-fit یا best-fit) که به دنبال پیدا کردن فضاهای خالی به اندازه کافی بزرگ برای تخصیص به برنامه‌ها هستند، بسیار مفید است. به عنوان مثال، اگر بدانیم ۴ بلاک متوالی آزاد داریم، می‌توانیم تصمیم بگیریم که آیا این فضا برای درخواست حافظه یک برنامه کافی است یا خیر.

۲/۱۰/۲ توضیح کد

کد C++ ارائه شده برای شمارش صفرهای متوالی در آرایه‌ای از بایت‌ها (`uint8_t`) به این صورت عمل می‌کند:

تابع `countZeros`

این تابع دو ورودی می‌گیرد: `arr` که اشاره‌گری به ابتدای آرایه بایت‌ها است و `len` که طول آرایه را مشخص می‌کند.

یک `<vector<int>` به نام `counts` تعریف می‌کند که برای ذخیره تعداد صفرهای متوالی استفاده می‌شود.

ابتدا بررسی می‌کند که آیا آرایه `nullptr` است یا طول آن صفر یا منفی است. در این صورت، یک `vector` خالی برمی‌گرداند.

حلقه بیرونی (`for (int i = 0; i < len; ++i)`): این حلقه روی تک تک بیت‌های آرایه ورودی تکرار می‌شود. متغیر `i` اندیس بایت فعلی را مشخص می‌کند.

حلقه داخلی (`for (int j = 7; j >= 0; --j)`): این حلقه روی تک تک بیت‌های بایت فعلی (از بیت ۷ یا با ارزش‌ترین بیت تا بیت ۰ یا کم ارزش‌ترین بیت) تکرار می‌شود. متغیر `j` اندیس بیت فعلی را مشخص می‌کند.

بررسی بیت با استفاده از شیفت بیتی (`(byte & (1 << j)) == 0`): این قسمت مهم‌ترین بخش کد است. عبارت `(j << 1)` یک عدد با پنري ایجاد می‌کند که فقط بیت زیم آن ۱ است و بقیه بیت‌ها ۰ هستند. به عنوان مثال، اگر `j` برابر ۳ باشد، `(3 << 1)` برابر `0b00001000` خواهد بود. سپس عملگر `&` (AND بیتی) بین این عدد و بایت فعلی (byte) انجام می‌شود. نتیجه این عمل فقط در بیت زیم غیر صفر خواهد بود اگر بیت زیم byte نیز ۱ باشد. در غیر این صورت (اگر بیت زیم byte صفر باشد)، نتیجه ۰ خواهد بود. شرط بررسی می‌کند که آیا نتیجه عمل AND بیتی ۰ است یا خیر. اگر ۰ باشد، یعنی بیت زیم byte صفر بوده و `currentCount` (شمارنده صفرهای متوالی) یک واحد افزایش می‌یابد.

ذخیره تعداد صفرهای متوالی: اگر بیت زیم ۱ باشد (یعنی به انتهای یک دنباله از صفرها رسیده‌ایم)، شرط `> 0` بررسی می‌شود. اگر `currentCount` بزرگتر از ۰ باشد (یعنی حداقل یک صفر متوالی وجود داشته باشد)، مقدار `currentCount` به `currentCount` اضافه می‌شود و سپس `vector counts` بازنشانی می‌شود تا شمارش دنباله بعدی صفرها شروع شود.

ذخیره صفرهای انتهایی: بعد از اتمام حلقه‌های داخلی و بیرونی، ممکن است `currentCount` همچنان بزرگتر از ۰ باشد (اگر دنباله صفرها در انتهای آرایه باشد). در این صورت، مقدار `currentCount` نیز به `vector counts` اضافه می‌شود.

تابع `main`: در تابع `main` می‌مثال با آرایه ۵ خانه‌ای تعریف شده و تابع `countZeros` با آنها فراخوانی می‌شود. سپس نتایج چاپ می‌شوند.

```

1  /* count Consecutive Zeros
2   * @author Shahriar
3   * @date Jan 1, 2025
4   */
5 #include <iostream>
6 #include <vector>
7 using namespace std;
8
9 vector<int> countZeros(uint8_t* arr, int len) {
10    vector<int> counts;
11    if (arr == nullptr || len <= 0) {
12        return counts;
13    }
14    int currentCount = 0;
15    for (int i = 0; i < len; ++i) {
16        uint8_t byte = arr[i];
17        for (int j = 7; j >= 0; --j) {
18            if ((byte & (1 << j)) == 0)
19                currentCount++;
20            else if (currentCount > 0) {
21                counts.push_back(currentCount);
22                currentCount = 0;
23            }
24        }
25    }
26    if (currentCount > 0)
27        counts.push_back(currentCount);
28 }
29 return counts;
30 }
31
32 int main() {
33     uint8_t arr[5];
34     arr[0] = 0b10011100;
35     arr[1] = 0b10000111;
36     arr[2] = 0b01000110;
37     arr[3] = 0b00000001;
38     arr[4] = 0b11111011;
39     vector<int> zeroCounts = countZeros(arr, 5);
40     cout << "Zeros: ";
41     for (int count : zeroCounts) {
42         cout << count << ", ";
43     }
44     cout << endl;
45     return 0;
46 }

```

شکل ۱۵- کد شمارش صفرهای متوالی

با توجه به نمونه کد بالا در تابع main خروجی بصورت زیر خواهد بود:

```

1 PS D:\Files\Word\OS\Extra\output> & .\countZero.exe
2 Zeros: 2, 2, 4, 1, 3, 8, 1,

```

شکل ۱۶- خروجی تابع صفرهای متوالی

۲/۱۱-۹-الگوریتم جایگزینی

حداقل ۴ الگوریتم جایگزینی صفحه را پیاده‌سازی کنید.

۲/۱۱/۱ پیاده‌سازی الگوریتم FIFO

الگوریتم FIFO (First-In, First-Out) یکی از ساده‌ترین الگوریتم‌های جایگزینی صفحه در حافظه مجازی است. در این الگوریتم، صفحه‌ای که از همه زودتر وارد حافظه شده، اولین صفحه‌ای است که در صورت نیاز به جایگزینی، از حافظه خارج می‌شود.

```

● ● ●
1  /* FIFO Page Replacement
2  * @author Shahriar
3  * @date Jan 1, 2025
4  */
5  package tamrin9_1;
6  import java.util.LinkedList;
7  import java.util.Queue;
8
9  public class JavaApp {
10     public static int pageFaults(int[] pages, int capacity) {
11         Queue<Integer> memory = new LinkedList<>();
12         java.util.Set<Integer> pageSet = new java.util.HashSet<>();
13         int pageFaultCount = 0;
14         for (int page : pages) {
15             if (!pageSet.contains(page)) {
16                 pageFaultCount++;
17                 if (memory.size() == capacity) {
18                     int oldestPage = memory.poll();
19                     pageSet.remove(oldestPage);
20                 }
21                 memory.offer(page);
22                 pageSet.add(page);
23             }
24         }
25         return pageFaultCount;
26     }
27
28     public static void main(String[] args) {
29         int[] pages = {1, 3, 0, 3, 5, 6, 3, 7, 8, 2, 1, 4, 9, 1};
30         int capacity = 5;
31         int faults = pageFaults(pages, capacity);
32         System.out.println("Page Fault Count: " + faults);
33     }
34 }
35

```

شکل ۱۷- الگوریتم جایگزینی صفحه FIFO

```

● ● ●
1 PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin9_1.JavaApp'
2 Page Fault Count: 11

```

شکل ۱۸- خروجی الگوریتم جایگزینی صفحه FIFO

۲/۱۱/۲ پیاده‌سازی الگوریتم LRU

الگوریتم LRU (Least Recently Used) یکی از الگوریتم‌های پرکاربرد جایگزینی صفحه در سیستم‌های مدیریت حافظه است. این الگوریتم صفحه‌ای را برای جایگزینی انتخاب می‌کند که کمترین زمان استفاده را در گذشته داشته است. پیاده‌سازی LRU با استفاده از ماتریس $n \times n$ (که n تعداد فریم‌های حافظه است) یکی از روش‌های موجود است که درک آن نسبتاً آسان است، اما به دلیل سربار زیاد برای تعداد فریم‌های بالا، در عمل کمتر مورد استفاده قرار می‌گیرد.

در اینجا پیاده‌سازی الگوریتم LRU با روش ماتریس $n \times n^*$ در زبان جاوا به همراه توضیحات کامل ارائه می‌شود:

```

1  /* LRU Page Replacement
2   * @author Shahriar
3   * @date Jan 1, 2025
4   */
5 package tamrin9_2;
6 import java.util.Arrays;
7
8 public class JavaApp {
9     public static int pageFaults(int[] pages, int capacity) {
10         int pageFaultCount = 0;
11         int[][] lruMatrix = new int[capacity][capacity];
12         int[] frames = new int[capacity];
13         Arrays.fill(frames, -1);
14         for (int page : pages) {
15             boolean pageHit = false;
16             int frameIndex = -1;
17             for (int i = 0; i < capacity; i++) {
18                 if (frames[i] == page) {
19                     pageHit = true;
20                     frameIndex = i;
21                     break;
22                 }
23             }
24             if (pageHit)
25                 updateLRUMatrix(lruMatrix, frameIndex, capacity);
26             else {
27                 pageFaultCount++;
28                 int lruIndex = findLRUIndex(lruMatrix, capacity);
29                 frames[lruIndex] = page;
30                 resetLRUMatrixRowAndColumn(lruMatrix, lruIndex, capacity);
31                 updateLRUMatrix(lruMatrix, lruIndex, capacity);
32             }
33         }
34         return pageFaultCount;
35     }
36
37     private static void updateLRUMatrix(int[][] matrix, int index, int capacity) {
38         for (int i = 0; i < capacity; i++) {
39             if (i != index) {
40                 matrix[index][i] = 1;
41                 matrix[i][index] = 0;
42             }
43         }
44     }
45
46     private static void resetLRUMatrixRowAndColumn(int[][] matrix, int index, int capacity) {
47         for (int i = 0; i < capacity; i++) {
48             matrix[index][i] = 0;
49             matrix[i][index] = 0;
50         }
51     }
52
53     private static int findLRUIndex(int[][] matrix, int capacity) {
54         for (int i = 0; i < capacity; i++) {
55             boolean isLRU = true;
56             for (int j = 0; j < capacity; j++) {
57                 if (matrix[j][i] == 1) {
58                     isLRU = false;
59                     break;
60                 }
61             }
62             if (isLRU) {
63                 return i;
64             }
65         }
66         return -1;
67     }
68
69     public static void main(String[] args) {
70         int[] pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
71         int capacity = 8;
72         int faults = pageFaults(pages, capacity);
73         System.out.println("Page Fault Count LRU: " + faults);
74     }
75 }
76

```

شکل ۱۹ - پیاده‌سازی الگوریتم LRU

```
PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin9_2.JavaApp'
2 Page Fault Count LRU: 20
```

شکل ۲۰- خروجی الگوریتم جایگزینی صفحه LRU

۲/۱۱/۳ پیاده‌سازی الگوریتم MFU

الگوریتم MFU (Most Frequently Used) یا بیشترین استفاده شده، بر خلاف LRU، صفحه‌ای را برای جایگزینی انتخاب می‌کند که بیشترین تعداد ارجاع را داشته است. فرض بر این است که صفحه‌ای که بیشترین استفاده را داشته، احتمالاً در آینده نیز مورد استفاده قرار خواهد گرفت. این الگوریتم در عمل کمتر از FIFO یا LRU استفاده می‌شود، زیرا عموماً عملکرد خوبی ندارد و پیاده‌سازی آن نیز نسبتاً پیچیده است.

در اینجا یک پیاده‌سازی از الگوریتم MFU در جاوا به همراه توضیحات کامل ارائه می‌شود:

```
/*
 * MFU Page Replacement
 * @author Shahriar
 * @date Jan 1, 2025
 */
package tamrin9_3;
import java.util.HashMap;
import java.util.Map;

public class JavaApp {
    public static int pageFaults(int[] pages, int capacity) {
        int pageFaultCount = 0;
        Map<Integer, Integer> frameUsage = new HashMap<>();
        java.util.Set<Integer> pageSet = new java.util.HashSet<>();
        for (int page : pages) {
            if (pageSet.contains(page)) {
                frameUsage.put(page, frameUsage.get(page) + 1);
            } else {
                pageFaultCount++;
                if (frameUsage.size() == capacity) {
                    int mfuPage = findMFUPage(frameUsage);
                    frameUsage.remove(mfuPage);
                    pageSet.remove(mfuPage);
                }
                frameUsage.put(page, 1);
                pageSet.add(page);
            }
        }
        return pageFaultCount;
    }

    private static int findMFUPage(Map<Integer, Integer> frameUsage) {
        int mfuPage = -1;
        int maxUsage = -1;
        for (Map.Entry<Integer, Integer> entry : frameUsage.entrySet()) {
            if (entry.getValue() > maxUsage) {
                maxUsage = entry.getValue();
                mfuPage = entry.getKey();
            }
        }
        return mfuPage;
    }

    public static void main(String[] args) {
        int[] pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
        int capacity = 8;
        int faults = pageFaults(pages, capacity);
        System.out.println("Page Fault Count MFU: " + faults);
    }
}
```

شکل ۲۱- پیاده‌سازی الگوریتم جایگزینی صفحه MFU



```
PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin9_3.JavaApp'
2 Page Fault Count MFU: 6
```

شکل ۲۲ - خروجی الگوریتم جایگزینی صفحه MFU

۴/۱۱/۴ پیاده‌سازی الگوریتم ساعت

الگوریتم جایگزینی صفحه به روش ساعت (Clock Algorithm) یک الگوریتم نسبتاً ساده و کارآمد برای مدیریت حافظه مجازی است. این الگوریتم بهبودیافته‌ای از الگوریتم FIFO است و سعی می‌کند با استفاده از یک بیت مرتع (Reference Bit)، عملکرد بهتری نسبت به FIFO داشته باشد.

در اینجا پیاده‌سازی الگوریتم ساعت به زبان جاوا به همراه توضیحات کامل ارائه می‌شود:

```

1  /* Clock Algorithm Page Replacement
2   * @author Shahriar
3   * @date Jan 1, 2025
4   */
5 package tamrin9_4;
6 import java.util.ArrayList;
7 import java.util.List;
8
9 public class JavaApp {
10    public static int pageFaults(int[] pages, int capacity) {
11        int pageFaultCount = 0;
12        List<Page> frames = new ArrayList<>(capacity);
13        int clockHand = 0;
14        for (int i = 0; i < capacity; i++) {
15            frames.add(new Page(-1));
16        }
17        for (int page : pages) {
18            boolean pageHit = false;
19            for (Page frame : frames) {
20                if (frame.number == page) {
21                    pageHit = true;
22                    frame.referenceBit = 1;
23                    break;
24                }
25            }
26            if (!pageHit) {
27                pageFaultCount++;
28                while (true) {
29                    Page currentPage = frames.get(clockHand);
30                    if (currentPage.referenceBit == 0) {
31                        currentPage.number = page;
32                        currentPage.referenceBit = 1;
33                        clockHand = (clockHand + 1) % capacity;
34                        break;
35                    } else {
36                        currentPage.referenceBit = 0;
37                        clockHand = (clockHand + 1) % capacity;
38                    }
39                }
40            }
41        }
42        return pageFaultCount;
43    }
44
45    private static class Page {
46        int number;
47        int referenceBit;
48
49        public Page(int number) {
50            this.number = number;
51            this.referenceBit = 0;
52        }
53    }
54
55    public static void main(String[] args) {
56        int[] pages = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
57        int capacity = 8;
58        int faults = pageFaults(pages, capacity);
59        System.out.println("Page Fault Count Clock Algorithm: " + faults);
60    }
61 }
62

```

شکل ۲۳- پیاده‌سازی الگوریتم جایگزینی صفحه ساعت

```

1 PS D:\Files\Word\OS> d;; cd 'd:\Files\Word\OS'; & 'C:\Program Files\Java\jdk-1.8\bin\java.exe' 'tamrin9_4.JavaApp'
2 Page Fault Count Clock Algorithm: 6

```

شکل ۲۴- خروجی الگوریتم جایگزینی صفحه ساعت

۲/۱۲ تمرین اضافه - دسترسی فایل در لینوکس

در سیستم‌عامل‌ها، سطوح دسترسی فایل‌ها مکانیسمی برای کنترل چگونگی دسترسی کاربران و برنامه‌ها به فایل‌ها و پوشش‌ها هستند. این سطوح تبیین می‌کنند که هر کاربر یا گروه کاربری چه عملیاتی را می‌تواند روی یک فایل انجام دهد. سه نوع دسترسی اصلی وجود دارد: خواندن (Read) که به کاربر اجازه می‌دهد محتوای فایل را ببیند، نوشتن (Write) که امکان تغییر و ذخیره فایل را فراهم می‌کند و اجرا (Execute) که اجازه اجرای فایل (اگر یک برنامه باشد) را می‌دهد. این دسترسی‌ها معمولاً برای سه دسته از کاربران تعریف می‌شوند: مالک فایل (Owner)، گروه مالک فایل (Group) و سایر کاربران (Others). به این ترتیب، می‌توان مشخص کرد که مثلاً فقط مالک فایل اجازه نوشتند داشته باشد، یا همه کاربران بتوانند فایل را بخوانند. این سیستم دسترسی، نقش مهمی در امنیت سیستم ایفا می‌کند و از دسترسی‌های غیرمجاز به فایل‌های حساس جلوگیری می‌کند.

دستور chmod در سیستم‌عامل‌های مبتنی بر یونیکس (مانند لینوکس) برای تغییر سطوح دسترسی فایل‌ها و دایرکتوری‌ها استفاده می‌شود. در لینوکس، هر فایل و پوشش دارای سه نوع سطح دسترسی اصلی است: خواندن (r)، نوشتن (w) و اجرا (x). این سطوح دسترسی برای سه دسته از کاربران تعریف می‌شوند: مالک فایل (u)، گروه مالک فایل (g) و سایر کاربران (o). به عنوان مثال، rwx به معنای دسترسی کامل (خواندن، نوشتن و اجرا) است. در حالی که r-- به معنای فقط دسترسی خواندن است. دستور chmod این اجازه را می‌دهد تا سطوح دسترسی را برای هر دسته از کاربران به صورت جداگانه تنظیم شوند.

نحوه کارکرد chmod به دو صورت عددی و نمادین است. در حالت عددی، هر سطح دسترسی با یک عدد مشخص می‌شود: ۴ برای خواندن، ۲ برای نوشتند و ۱ برای اجرا. سپس این اعداد برای هر دسته از کاربران با هم جمع می‌شوند. به عنوان مثال، ۷۵۵ به معنای rwxr-xr-- برای مالک، r-Xr-- برای گروه و r-Xr-- برای سایر کاربران است. در حالت نمادین، از حروف u, g, o و a (برای همه) و عملگرهای + (اضافه کردن)، - (حذف کردن) و = (تنظیم کردن) استفاده می‌شود. به عنوان مثال، chmod u+x filename دسترسی اجرا را به مالک فایل filename اضافه می‌کند. این دستور ابزاری قدرتمند برای مدیریت امنیت و کنترل دسترسی به فایل‌ها و پوشش‌ها در لینوکس است.

۱. روش عددی:

- دسترسی کامل (rwx) معادل عدد ۷ است.
- خواندن و اجرا (r-x) معادل عدد ۵ است.
- بنابراین، دسترسی مورد نظر ما به صورت عددی ۷۵۵ خواهد بود.

۲. روش نمادین:

- برای مالک (u) دسترسی خواندن، نوشتند و اجرا را اضافه می‌کنیم: u+rwx
- برای گروه (g) دسترسی خواندن و اجرا را اضافه می‌کنیم: g+rx
- برای سایرین (o) دسترسی خواندن و اجرا را اضافه می‌کنیم: o+rX

```
root@Shahriar: ~/Desktop/myFolder
root@Shahriar:~/Desktop/myFolder# chmod 755 myBash.sh
root@Shahriar:~/Desktop/myFolder# chmod u+rwx,g+rx,o+rX myBash.sh
root@Shahriar:~/Desktop/myFolder#
```

شکل ۲۵ - دستور chmod

۲/۱۳/۱۰ - بردار وقفه و DMA

بردارهای وقفه (Interrupt vectors) و روش دسترسی مستقیم انتقال داده DMA mode of data transfer (DMA) را بطور کامل توضیح دهید.

۲/۱۳/۱۱ بردارهای وقفه (Interrupt Vectors)

بردار وقفه به عنوان یکی از عناصر کلیدی در مدیریت وقفه‌ها، نقش بسیار مهمی در سیستم‌های کامپیووتری دارد. این بردارها شامل مجموعه‌ای از آدرس‌ها هستند که هر آدرس به یک روتین خاص از نوع سرویس وقفه (ISR)^۱ اشاره دارد. این ساختار به پردازنده کمک می‌کند که بتواند به طور سریع و کارآمد به انواع مختلف وقفه‌ها پاسخ دهد. هر وقفه‌ای که در سیستم رخ می‌دهد (مانند درخواست I/O یا خطای تقسیم بر صفر)، یک شماره یا اندیس دارد که برای جستجوی آدرس مربوطه در بردار وقفه استفاده می‌شود.

یکی از نکات کلیدی در استفاده از بردار وقفه این است که پردازنده باید بتواند به سرعت منبع وقفه را شناسایی کند و فرآیند فعلی را متوقف کند تا به روتین مربوطه برود. برای مثال، در یک سیستم با چهار نوع وقفه (System Call, Trap, و وقفه I/O، و وقفه نرمافزاری)، هر کدام از این وقفه‌ها یک اندیس خاص دارند که از آن برای دسترسی به آدرس ISR مربوطه استفاده می‌شود. پردازنده از این اندیس برای جستجو در آرایه بردار وقفه استفاده می‌کند و پس از یافتن آدرس ISR مربوطه، کنترل را به آن منتقل می‌کند.

سیستم‌های پیچیده‌تر ممکن است بردارهای وقفه‌ای داشته باشند که علاوه بر مدیریت وقفه‌ها، اولویت‌ها را نیز در نظر بگیرند. به این صورت که اگر دو یا چند وقفه به طور همزمان رخ دهند، بردار وقفه می‌تواند مشخص کند کدامیک از آن‌ها باید ابتدا مدیریت شود. این روش معمولاً در سیستم‌های بلادرنگ^۲ استفاده می‌شود، جایی که پاسخ‌دهی سریع به وقفه‌ها بحرانی است.

یکی از مهم‌ترین کاربردهای بردار وقفه، ساده‌سازی مدیریت وقفه‌ها و افزایش کارایی سیستم است. این مکانیزم به سیستم عامل اجازه می‌دهد که وقفه‌ها را به روشی ساخت‌یافته مدیریت کند و به راحتی ISRهای جدید را به سیستم اضافه یا تغییر دهد. علاوه بر این، این روش انعطاف‌پذیری بیشتری را در طراحی سیستم ایجاد می‌کند و امکان پیاده‌سازی سیستم‌هایی با وقفه‌های متعدد و پیچیده را فراهم می‌کند.

۲/۱۳/۱۲ روش انتقال داده به روش DMA

انتقال داده به روش DMA^۳ یک تکنیک پیشرفته در سیستم‌های کامپیووتری است که به دستگاه‌های ورودی/خروجی (I/O) اجازه می‌دهد داده‌ها را مستقیماً بین حافظه اصلی و دستگاه منتقل کنند، بدون اینکه نیاز به دخالت پردازنده باشد. این روش زمانی بسیار کارآمد است که نیاز به انتقال حجم بالایی از داده‌ها باشد، زیرا می‌تواند بار پردازشی پردازنده را کاهش دهد و کارایی کلی سیستم را افزایش دهد.

در فرآیند DMA، پردازنده در ابتدا عملیات را رام‌دانزی می‌کند و اطلاعاتی مانند آدرس شروع در حافظه، تعداد داده‌های انتقالی، و نوع انتقال را به کنترلر DMA می‌دهد. سپس کنترلر DMA، وظیفه انتقال داده‌ها را به عهده می‌گیرد. برای مثال، در انتقال داده از دیسک به حافظه، ابتدا داده‌ها وارد بافر کنترلر دیسک می‌شوند. کنترلر DMA سپس از طریق گذرگاه حافظه داده‌ها را به صورت مستقیم به حافظه اصلی منتقل می‌کند، بدون اینکه پردازنده درگیر شود.

یکی از مزایای کلیدی این روش، توانایی آن در مدیریت انتقال داده به صورت مستقل از پردازنده است. به عبارت دیگر، کنترلر DMA می‌تواند چرخه‌های گذرگاه حافظه را سرقت کند و داده‌ها را انتقال دهد، در حالی که پردازنده به اجرای برنامه‌های دیگر ادامه می‌دهد. این روش به ویژه برای سیستم‌هایی که نیاز به پردازش بلادرنگ دارند یا در آن‌ها ترافیک شبکه با حجم بالا رخ می‌دهد، بسیار مفید است. به عنوان مثال، کارت‌های شبکه معمولاً از DMA برای انتقال سریع داده‌ها استفاده می‌کنند.

Interrupt Service Routine^۴

Real-Time Systems^۵

Direct Memory Access^۶

از سوی دیگر، روش DMA معایبی نیز دارد. به عنوان مثال، نیاز به سختافزار پیچیده‌تر مانند کنترلر DMA دارد و همچنین می‌تواند باعث تداخل در دسترسی به گذرگاه حافظه شود. برای مدیریت این چالش‌ها، سیستم‌عامل باید به درستی تداخل‌ها را مدیریت کند و از مکانیزم‌هایی مانند زمان‌بندی دسترسی به حافظه استفاده کند. با این وجود، مزایای این روش مانند افزایش سرعت و کاهش بار پردازندۀ، اغلب بر معایب آن غلبه می‌کند.

۳ منابع

OPERATING SYSTEM CONCEPTS WITH JAVA (8TH EDITION): ABRAHAM SILBERSCHATZ, PETER B. GALVIN, GREG GAGNE PP. 1042 (2009) [ISBN:047050949X]

MODERN OPERATING SYSTEMS (3RD EDITION): ANDREW S. TANENBAUM PP. 1094 (2007) [ISBN:0136006639]

OPERATING SYSTEMS: INTERNALS AND DESIGN PRINCIPLES: WILLIAM STALLINGS PP. 840 (2008) [ISBN:0136006329]

A VISIT TO MUTUAL EXCLUSION IN SEVEN DATES: MICHEL RAYNAL; GADI TAUBENFELD IN THEORETICAL COMPUTER SCIENCE VOL. 919 PP. 47-65 (2022) [10.1016/J.TCS.2022.03.030]

WIKIPEDIA: OPERATION SYSTEM, MUTEX, DMA, ALGORITHM [WIKIPEDIA.ORG]

GEEKSFORGEEKS: BAKERY ALGORITHM, MEMORY MANAGEMENT [GEEKSFORGEEKS.ORG]

JAVAPPOINT: JAVA IN OPERATION SYSTEMS [JAVATPOINT.COM]