Ф. Воякин

И. Алексей

О. Янович

Студент ИУ7-14Б
(группа)

2018—2022 уч. г.

Начальник отдела охраны

А. Максименко

Выдан «4» 09 2018 г.

ИУ7-54Б Воякин Алексей

Билет 20.

№1

Процессы взаимодействие параллельных процессов - монопольный доступ и взаимоисключение; приоритетная реализация ч/з взаимоискл. - примеры, семафоры - определение, виды семафоров, примеры использования множественных семафоров из л.р. „производство - потребление" и „читатели - писатели".

Каждый процесс имеет собственное защищённое адресное пространство. Процессы не могут разделять глобальные переменные.

Системы предоставляют соответствующие системные вызовы, для того чтобы процессы могли взаимодействовать.

Прежде чем рассматривать IPC (Inter Process Communication) - общее название средств взаимодействия параллельных процессов, рассмотрим саму проблему.

Представим работу двух филиалов банка. В конце дня каждый из филиалов должен перечислить на определённый счёт S всю выручку.

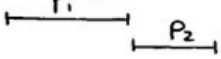
P_1: $S = S + B_1$; ①②③④⑤⑥⑦⑧

P_2: $S = S + B_2$; ①④⑤⑥⑦⑧

Квант получает первый процесс, считывает значения B_1 и S и складывает их, далее процесс P_2 перебивает P_1 и выполняет тоже действие, при этом считывает старое значение S. Далее квант получает первый процесс, но он уже сложил B_1 и S, таким образом наблюдаем потерю составляющей второго филиала.

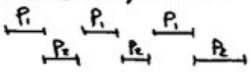
Рассмотрим уровни наблюдения.

1. Последовательное выполнение.



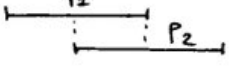
От начала до конца выполняется процесс P_1, потом от начала до конца выполняется процесс P_2.

2. Квазипараллельное выполнение.



Квант времени выполняются команды процесса P_1, потом происходит переключение и квант времени выполняются команды процесса P_2. В общем то, последовательное выполнение, но с точки зрения наблюдателя - параллельное.

3. Реально параллельное выполнение.



Совпадение по времени выполнения команд процесса P_1 и команд процесса P_2.

Рассмотрим два варианта (квазипараллельное и реально параллельное выполнение):

Оба процесса выполняют одну и ту же последовательность команд.

P_1:	P_2:
mov eax, myvar	mov eax, myvar
inc eax	inc eax
mov myvar, eax	mov myvar, eax.

Наши компьютеры имеют SMP- архитектуру (многоядерная архитектура, где каждое ядро это процессор со своими регистрами, в каждом ядре полный набор регистров)

Каждый процессор будет иметь свой регистр EAX. Все процессоры работают с одной памятью.

P_1 на проц 1	eax	myvar	eax	P_2 на проц. 2.
–	–	0	–	—
mov eax, myvar	0	0	–	—
–	0	0	0	mov eax, myvar
–	0	0	1	inc eax
–	0	1	1	mov myvar, eax
inc eax	1	1	1	–
mov myvar, eax	1	1	1	–

P_1 на проц. 1	eax / myvar	P_2 на проц. 2
	0	
	Вытесняется P_1	
mov eax, myvar	0 / 0	
P_2 вытеснен, его контекст сохранен		
	0 / 0	mov eax, myvar
	0 / 1	inc eax
	1 / 1	mov myvar, eax
P_2 вытеснен, его контекст сохранен		
inc eax	1 / 1	
mov myvar, eax	1 / 1	

В обоих случаях мы потеряли данные. Даже не смотря на квазипараллельность, мы видим что переключение связано с переключением контекста, с запоминанием содержимого регистров, соответственно мы получили тот же самый результат.

Подобные действия принято называть критическими, а часть кода в которой выполняются такие действия принято называть критической секцией.

Myvar – разделяемая переменная.

Для того чтобы не терять данные необходимо обеспечить монопольное использование разделяемых параллельными процессами ресурсов. Т.е каждый процесс должен владеть разделяемым ресурсом монопольно. Монопольный доступ к разделяемому ресурсу обеспечивается методами взаимоисключения. Т.е если один процесс находится в своей критической секции по некоторой переменной, то другой процесс не может войти в эту критическую секцию по той же переменной до тех пор пока первый процесс не освободит её.

Существует ряд способов реализации взаимоисключения:

1. Программный
2. Аппаратный.
3. С использованием семафоров.
4. С использованием мониторов.

Программный способ, примеры.

① Использование флага.

```
P1:
  while (1)
  begin
    while (flag P2); // цикл акт. ожид.
    flag P1 = 1;
    CR1; // крит. секция.
    flag P1 = 0;
    PR1;
  end;
```

```
P2:
  while (1)
  begin
    while (flag P1);
    flag P2 = 1;
    CR2;
    flag P2 = 0;
    PR2;
  end;
```

```
Объявления и нач. установки
flag P1, flag P2 : logical;
flag P1 = 0;
flag P2 = 0;
par begin
  P1 ; P2;
par end.
```

В P_1 входим в while по переменной flag P_2, он сброшен. P_1 теряет квант. Процесс P_2 выполняет тоже самое, входит в цикл проверки флага P_1. Флаг P_1 сброшен, устанавливает свой флаг, входит в крит. секцию. Выйдя из критической секции сбрасывает свой флаг. Квант опять получает процесс P_1, но проверка flag P_2 уже выполнена, соответственно устанавливает свой флаг в единицу, входит в крит. секцию и мы наблюдаем потерю значения.

② Установка флага своего перед проверкой флага другого процесса.

```
P1:
  while (1)
  begin
    flag P1 = 1;
    while (flag P2);
    CR1;
    flag P1 = 0;
    PR1;
  end;
```

```
P2:
  while (1)
  begin
    flag P2 = 1;
    while (flag P1);
    CR2;
    flag P2 = 0;
    PR2;
  end;
```

```
Объявления и нач. установки.
flag P1, flag P2: logical;
flag P1 = 0;
flag P2 = 0;
par begin
  P1 ; P2;
par end.
```

Оба процесса будут в цикле активного ожидания, тратя на это процессорное время. Ни один из процессов не может продолжить своё выполнение, они находятся в тупике (dead lock). Это классический пример тупиковой ситуации.

В системе имеются 3 негативные ситуации:

1. Бесконечное ожидание.
2. Тупиковая ситуация.
3. Захват и освобождение одних и тех же ресурсов.

③ Алгоритм Деккера.

Чисто программный способ, алг. исключает все негативные ситуации, является надёжным, искл. попадание процессов в тупик, искл. беск. откл.

Но данную задачу Деккер решил только для двух параллельных процессов.

Он ввел третью переменную, которую назвал - "чья очередь".

```
P1:
  While(1)
  begin
    flagP1=1;
    while(flagP2);
    begin
      if(que==2)then
      begin
        flagP1=0;
        while(que==2);
        flagP1=1;
      end;
    end;
    CR1;
    flagP1=0;
    que=2;
    PR1;
end;
```

```
P2:
  while(2)
  begin
    flagP2=1;
    while(flagP1);
    begin
      if(que==1)then
      begin
        flagP2=0;
        while(que==1);
        flagP2=1;
      end;
    end;
    CR2;
    flagP2=0;
    que=1;
    PR2;
end;
```

```
Объявления и нач. установки.
flagP1, flagP2 : logical;
que: int;
flagP1=0;
flagP2=0;
que=1;
par begin
  P1; P2;
par end.
```

Здесь мы видим следующее решение - введена дополнительная переменная "que", при этом мы видим, что прежде чем войти в цикл проверки флага другого процесса, процесс устанавливает свой флаг. Если флаг другого процесса установлен, то процесс проверяет переменную "que". Если очередь другого процесса, то процесс сбрасывает свой флаг и переходит в цикл активного ожидания по переменной "que". Выйдя из этого цикла, процесс устанавливает свой флаг и входит в свой крит. уч. Выйдя из крит. уч. процесс сбрасывает свой флаг и устанавливает, что следующим в крит. сек. войдёт другой процесс.

④ Алгоритм Лампорта (Lamport)

Это чисто программное решение. Алг. получил название "Булочная" ("Bakery")
Этот алг. решает проблему крит. секции для n прикладных процессов.

Основная ~~то~~ идея взята из работы булочной.
Потребители берут номера (устанавливается очередь), тот кто имеет наименьший номер обслуживается следующим (вхождение в крит. секцию)

Семафор - это неотрицательная защищённая переменная, над которой определены две неделимые операции
P(S) - пропустить и V(S) - освободить.

Семафоры бывают 3-ёх типов:

1. бинарные (принимают знач. 0 или 1)
2. считающие (принимают неотр. целые значения)
3. множественные (это массивы считающих семафоров (могут работать и как бинарные).
   Одной неделимой операцией можно изменить все или часть семафоров набора).

Примеры использования множественных семафоров из Л.Р.:

① "Производство-потребление":

```
struct sembuf semops_produce_begin[2] = {
    {SE, -1, 0},
    {SB, -1, 0}
};
struct sembuf semops_produce_end[2] = {
    {SB, 1, 0},
    {SF, 1, 0}
};
struct sembuf semops_consume_begin[2] = {
    {SF, -1, 0},
    {SB, -1, 0}
};
struct sembuf semops_consume_end[2] = {
    {SB, 1, 0},
    {SE, 1, 0}
};
```

```
int get_sem_fd()
{
  int sem_fd = semget(100, 3, IPC_CREAT|permissions);
  if (sem_fd == -1)
  {
    perror("Semget failed. \n");
    exit(3);
  }
  return sem_fd;
}

int produce(int sem_fd, int number)
{
  if(semop(sem_fd, semops_produse_begin, 2) == -1)
  {
    perror("Semop failed. \n");
    exit(4);
  }
  int** shm_prod_pos = shm_buff - 3;
  int * prod_pos = *shm_prod_pos;
  if (*(prod_pos* -1) >= buffer_size)
  {
    if(semop(sem_fd, semops_produce_end, 2) == -1)
    {
      perror("Semop failed. \n");
      exit(5);
    }
    return 0;
  }
  *prod_pos = *(prod_pos - 1) + 1;
  printf("Производитель %d, pid = %d создал значение %d\n", number, getpid(), *prod_pos);
  prod_pos++;
  *shm_prod_pos = prod_pos;
  if (semop(sem_fd, semops_produce_end, 2) == -1)
  {
    perror("Semop failed. \n");
    exit(6);
  }
  return 0;
{

int consume(int sem_fd, int number)
{
  if (semop(sem_fd, consume_begin, 2) == -1)
  {
    perror("Semop failed. \n");
    exit(7);
  }
```

```
int **shm_cons_pos = shm_buffer -5;
int *cons_pos = *shm_cons_pos;
if (cons_pos - shm_buffer >= buffer_size)
{
  if (semop(sem_fd, semops_consume_end, 2) == -1)
  {
    perror("Semop failed.\n");
    exit(1);
  }
  return 0;
}
printf("Потребитель %d, pid=%d прочитал значение: %d\n", number, getpid(), *cons_pos);
cons_pos++;
*shm_cons_pos = cons_pos;
if (semop(sem_fd, semops_consume_end, 2) == -1)
{
  perror("Semop failed.\n");
  exit(8);
}
return 0;
}
```

② Читатели - писатели:

```
struct sembuf start_read[5] = {
    {Waiting_readers, 1, SEM_UNDO},
    {Waiting_writers, 0, SEM_UNDO},
    {Activewriter, 0, SEM_UNDO},
    {Waiting_readers, -1, SEM_UNDO},
    {Active_readers, 1, SEM_UNDO}
};
struct sembuf stop_read[1]={{Active_readers, -1, SEM_UNDO}};
struct sembuf start_write[5] = {
    {Waiting_writers, 1, SEM_UNDO},
    {Active_writer, 0, SEM_UNDO},
    {Active_readers, 0, SEM_UNDO},
    {Waiting_writers, -1, SEM_UNDO},
    {Active_writer, 1, SEM_UNDO}};
Struct sembuf stop_write[1] = {{Active_writer, -1, SEM_UNDO}};

void writer(const int semid, const int index)
{
```

```
    if (semop(semid, start_write, 5) == -1)
    {
      perror("Semop failed.\n");
      exit(2);
    }
    (*shm)++;
    printf("Писатель %d написал %d\n", index+1, *shm);
    if (semop(semid, stop_write, 1) == -1)
    {
      perror("Semop failed\n");
      exit(2);
    }
}
void reader(const int semid, const int index)
{
    if (semop(semid, start_read, 5) == -1)
    {
      perror("Semop failed.\n");
      exit(1);
    {
    printf("Читатель %d прочитал %d\n", index - WRITERS_NUM+1, *shm);
    if (semop(semid, stop_read, 1) == -1)
    {
      perror("Semop failed.\n");
      exit(1);
    }
}
```

№2.

Пересчёт динамических приоритетов в ОС UNIX и Windows (Л.Р).

Windows

При запуске процесса ему назначается приоритет, обычно он называется базовым приоритетом процесса. Приоритет потока вычисляется относительно базового приоритета процесса.

В Windows реализуется приоритетная, вытесняющая система планирования.

В системе 32 приоритета от 0 до 31.

- от 16 до 31 – уровни реального времени.
- от 0 до 15 – динамические уровни, 0 зарезервирован для потока обнуления стр.

Текущий приоритет потока в динамическом диапазоне может быть повышен:

- Вследствие событий планировщика или диспетчера. (сокр. задержек)
- Повышения приоритета, связанные с завершением ожидания.
- Повышение приоритета владельца блокировки.
- Повышение вследствие завершения ввода - вывода (сокр. задержек)
- Повышение при ожидании ресурсов исполняющей системы.
- Повышение приоритетов потоков первого плана после ожидания.
- Повышение приоритета после пробуждения GUI - потока.
- Повышения приоритета, связанные с перегруженностью Ц.П.
- Повышения приоритетов для мультимедийных приложений и игр.

Windows никогда не повышает приоритет потоков в диапазоне реального времени (16-31)

Текущий приоритет потока в динамическом диапазоне может быть понижен до базового приоритета путём вычитания всех повышений.

UNIX

Планирование проц. в UNIX основано на приоритете процесса. В современных UNIX ядро является вытесняющим - процесс в режиме ядра может быть вытеснен более приоритетным процессам в режиме ядра. Ядро было сделано вытесняющим для того, чтобы система могла обслуживать процессы реального времени (аудио, видео).

Приоритет задаётся целым числом от 0 до 127. Чем меньше число, тем выше приоритет. От 0 до 49 зарезервированы для ядра. Прикл. процессы в диапазоне от 50 до 127.

Структура дескриптора процесса proc содержит следующие поля, относящиеся к приоритетам:

- p_pri - текущий приоритет планирования
- p_usrpri - приоритет режима задачи.
- p_cpu - результат последнего измерения использования процессора.
- p_nice - фактор „любезности", устанавливаемый пользователем.

Приоритет в реж. задачи зависит:

- от фактора „любезности" (nice). Число от 0 до 39 со значением 20 по ум. Увеличение значения приводит к уменьшению приоритета. Фоновым процессам автоматически задаются более высокие значения.

- Степени загруженности процессора в момент последнего обслуживания
или процесса.

Системы с динамическими приоритетами пытаются выделить процессорное время так, чтобы конкурирующие процессы получили его в примерно равных количествах.

Каждую секунду ядро вызывает процедуру schedcpu(), которая уменьшает значения p_cpu каждого процесса исходя из decay factor.

$$decay = \frac{2 \cdot load\_average}{2 \cdot load\_average + 1}$$

load_average - это среднее кол-во процессов, находящихся в состоянии готовности к выполнению за пол. сек.

Процедура schedcpu() также пересчитывает приоритеты для режима задачи всех процессов по формуле.

$$P\_USRpri = PUSER + \frac{P\_cpu}{4} + 2 * p\_nice$$

где PUSER – базовый приоритет в режиме задачи, равный 50.

Чем дольше процесс простаивает в очереди на выполнение, тем больше фактор полураспада уменьшает его p_cpu, что приводит к повышению его приоритета. Такая схема предотвращает бесконечное откладывание низкоприоритетных процессов.