

кафедра штучного інтелекту та програмного забезпечення
факультету комп'ютерних наук
Харківського національного університету ім. В.Н. Каразіна

А. Горбань

Синхронізація процесів

Частина 1

Взаємовиключення, семафори та монітори

методичний посібник до курсу
"Операційні системи"

редакція Др130920

Харків 2020

ПРЕДИСЛОВИЕ

О данном документе. Настоящий документ является методическим пособием к курсу "Операционные системы", читаемому автором студентам факультета компьютерных наук ХНУ и, строго говоря, не является ни учебником ни конспектом лекций. Его основное назначение – представление набора средств и методов, использующихся для решения задач синхронизации процессов в современных вычислительных системах как на уровне общих понятий, так и на примерах их реализаций в конкретных ОС.

О его пользе. Автор не принимает на себя никакой ответственности за возможные последствия прочтения или непрочтения данного документа (касается, в основном, студентов ФКН ХНУ).

О стиле и обозначениях. "П" – далее везде по тексту читать как "процесс", "ОС" – "операционная система", "В/В" – "ввод/вывод", остальные сокращения расширяются по тексту. В распечатках (листингах) примеров комментарии зеленого цвета, черный цвет комментария подразумевает его замену соответствующим программным кодом.

Тексты примеров не на языке `c` а в стиле `c!` Основное отличие – передача параметров по ссылке.

Об источниках. Документ в заметной части является компилятивным. Заимствованные фрагменты, как правило, имеют соответствующие ссылки на источники. Список использованных литературных источников ограничен соображениями их доступности. Книги [1] и [2] (см. список литературы) являются рекомендованными (для студентов ФКН знакомство с ними еще до сессии крайне рекомендовано).

Об ограничениях. Документ не предназначен для использования в качестве шпаргалки. Поэтому запрещается его прямая печать либо изготовление твердых копий любым способом в форматах отличных от А4. Однако нет никаких ограничений на копирование и распространение исходного документа в формате PDF в целом.

И напоследок. Следите за номером редакции документа
(формат: Д <р|у> <ГГ> <ММ> <ДД>).

Настоящая редакция Др190920 отличается от редакций Др060410 и Др130909 в основном техническими правками.

Автор будет признателен за ваши отзывы и замечания, присланные на a.gorban@karazin.ua.

Читайте по 1 странице 3 раза в день перед сном – должно помочь.

А. Горбань

Особенности параллельных вычислений в однопроцессорной системе.

Следствием реализации многозадачного режима даже в однопроцессорной системе являются следующие осложнения:

1. Разделение **глобальных ресурсов** чревато опасностями. Например, чтение/запись глобальной переменной двумя ТП показывает важность порядка доступа.
2. ОС трудно управлять распределением ресурсов оптимально. Напр., ТП затребовал и получил канал В/В, а затем приостановил свою работу. Нежелательно чтобы ОС при этом продолжала удерживать канал за ТП.
3. Становится очень трудно обнаружить программную ошибку, т.к. результат работы программы перестает быть детерминированным и воспроизводимым.

Пример опасного использования глобальных переменных [1]: Листинг 1

```
int chin, chout; /* chin и chout - глобальные! */
void echo(void) {
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Пусть в однопользовательской многозадачной системе данная процедура является разделяемой всеми ТП. При возможна следующая последовательность событий:

1. Процесс P1 вызывает echo и прерывается по выполнении getchar
2. Активируется процесс P2, вызывает echo и выполняет ее до конца.
3. Продолжается выполнение P1. В результате первый символ теряется, а второй печатается дважды. Эти ситуации наз. **состояниями состязания** (race condition).

Предположим теперь, что выполнять процедуру echo одновременно ТП не могут. Тогда:

1. P1 вызывает echo и прерывается по выполнении getchar
2. Активируется P2, вызывает echo и поскольку он заблокирован от входа в нее, P2 приостанавливается до освобождения echo.
3. Продолжается выполнение P1, кот. завершает выполнение echo.
4. После выхода P1 из echo блокировка P2 снимается и он при возобновлении работы успешно выполняет echo.

Вывод: разделяемые глобальные переменные (как и другие разделяемые глобальные ресурсы) нуждаются в защите. Если хорошо подумать, то единственный способ сделать это - управлять кодом, осуществляющим доступ к этим переменным.

Заметьте, это следствие того, что в однопроцессорной системе прерывание может приостановить ТП в любой точке программы, а в многопроцессорной, к тому же, два ТП могут одновременно обратиться к одной глобальной переменной.

Для дальнейшего изложения существенно, что способы взаимодействия ТП можно классифицировать по степени осведомленности друг о друге:

- ТП не осведомлены о наличии друг друга. Наблюдается **конкуренция** за обладанием разделяемым ресурсом.
- ТП косвенно осведомлены о наличии друг друга. Такие ТП демонстрируют **сотрудничество при разделении общего объекта**.



- П непосредственно осведомлены о наличии друг друга. Они способны общаться с использованием идентификатора П и изначально созданы для совместной работы. Такие П демонстрируют **сотрудничество** при работе.

Таблица 1.

Взаимодействие процессов.

Степень осведомленности	Вид взаимодействия	Влияние одного П на другой	Потенциальные проблемы
П не осведомлены друг о друге	Конкуренция	<ul style="list-style-type: none"> · Результат работы одного П не зависит от действий других. · Возможно влияние работы одного П на время работы другого. 	<ul style="list-style-type: none"> · Необходимость взаимoisключений · Взаимоблокировки · Голодание
П косвенно осведомлены о наличии друг друга.	Сотрудничество с использованием разделения ресурсов	<ul style="list-style-type: none"> · Результат работы одного П может зависеть от информации, полученной от других П (через разделяемые ресурсы). · Возможно влияние работы одного П на время работы другого. 	<ul style="list-style-type: none"> · Необходимость взаимoisключений · Взаимоблокировки · Голодание
П непосредственно осведомлены о наличии друг друга (знают имена).	Сотрудничество с использованием связи	<ul style="list-style-type: none"> · Результат работы одного П может зависеть от информации (сообщений), полученной от других П. · Возможно влияние работы одного П на время работы другого. 	<ul style="list-style-type: none"> · Взаимоблокировки · Голодание

В случае конкуренции П мы сталкиваемся с тремя проблемами:

- ☛ **Необходимость взаимных исключений (mutual exclusion)** - т.е. исключения П, получившим доступ к разделяемому ресурсу, возможности одновременного доступа к этому ресурсу для всех остальных П. При этом такой ресурс наз. **критическим ресурсом**, а часть программы, которая его использует, наз. **критическим разделом (секцией) (critical section)**. Крайне важно, чтобы в критическом разделе в любой момент могла находиться только одна программа. Осуществление взаимных исключений создает две дополнительные проблемы:
- ☛ **Взаимная блокировка (deadlock)** . Напр. два П P1 и P2 нуждаются для исполнения в 2-х (одних и тех же) ресурсах R1 и R2 одновременно. При этом каждый из них удерживает по одному ресурсу и безуспешно пытается захватить второй.
- ☛ **Голодание**. Пусть три П P1, P2, P3 периодически нуждаются в одном и том же ресурсе. Теоретически возможна ситуация при которой доступ к ресурсу будут получать P1 и P2 в порядке очередности, а P3 никогда не получит доступа к нему, хотя никакой взаимной блокировки нет.

В случае сотрудничества П, не будучи осведомлены явно о наличии других П, тем не менее принимают меры к поддержанию целостности данных.

При сотрудничестве с использованием связи ПП принимают участие в общей работе, которая их и объединяет. Связь обеспечивает возможность синхронизации или координации действий ПП. Обычно можно считать, что связь состоит из посылки сообщений определенного типа. Прimitives для отправки и получения сообщений предоставляются языком программирования или ядром ОС. Поскольку при передаче сообщений не используются совместно какие-либо ресурсы, взаимноисключения не требуются. Однако проблемы взаимоблокировок и голодания остаются. Так, например, два ПП могут заблокировать друг друга взаимным ожиданием сообщений один от другого.

Любая возможность поддержки взаимных исключений должна соответствовать таким требованиям:

1. Взаимоисключения должны осуществляться в принудительном порядке.
2. Процесс, завершающий работу вне критического раздела, не должен влиять на другие ПП.
3. Не должна возникать ситуация бесконечного ожидания входа в критический раздел (исключение взаимоблокировок и голодания).
4. Когда в критическом разделе нет ни одного ПП, любой ПП, запросивший возможность входа в него, должен немедленно ее получить.
5. Не делается никаких предположений о количестве ПП или их относительных скоростях работы.
6. ПП остается в критическом разделе только в течение ограниченного времени.

Есть ряд подходов к реализации этих требований:

- Передача ответственности за соответствие требованиям самому ПП (программный подход).
- Использование машинных команд специального назначения.
- Предоставление опред. ур-ня поддержки со стороны языка программирования или ОС.

Программный подход к реализации взаимноисключений.

Напишем первый вариант взаимноисключения скажем так:

Листинг 2

```
boolean flag = false; /* переменная блокировки */

/* Процесс 1 */
while(flag); /*АКТИВНОЕ ОЖИДАНИЕ*/
flag = true;
/* КРИТИЧЕСКИЙ РАЗДЕЛ */
flag = false;
/* ОСТАЛЬНОЙ КОД */

/* Процесс 2 */
while(flag);
flag = true;
/* КРИТИЧЕСКИЙ РАЗДЕЛ */
flag = false;
/* ОСТАЛЬНОЙ КОД */
```

Здесь `flag` - глобальная переменная, которая управляет доступом к критическому разделу для всех П. Такие переменные наз. **переменными блокировки**. Недостаток алгоритмов такого типа - наличие состояния **активного ожидания**, т.е. ситуации когда в ожидании возможности входа в критический раздел П бесцельно потребляет процессорное время (и занимает системную шину), выполняя постоянные проверки состояния переменной блокировки. Блокировка, использующая активное ожидание, называется **спин-блокировкой** (spinlock).

Почему приведенный пример (лист. 2) большую часть времени будет работать верно, а меньшую - убивать разделяемые ресурсы?

Как поведет себя эта программа в зависимости от типа многозадачности среды исполнения - **кооперативной** (cooperative multitasking) или **вытесняющей** (preemptive m.)?

Первым удовлетворительное программное решение задачи взаимного исключения для двух процессов получил датский математик Деккер (Т. Dekker). Во всяком случае, Эдсгер Дейкстра (E. Dijkstra) именно на него ссылается в своей работе 1965 г. "Cooperating Sequential Processes". Подробно с алгоритмом Деккера можно ознакомиться в [1].

Более простой алгоритм был предложен Петерсоном (G.L. Peterson) в 1981 г. Ниже приведен текст программы с краткими комментариями для двух П (алгоритм легко обобщить на случай произвольного числа П).

Алгоритм Петерсона.

Листинг 3

```
boolean flag[2]; /* Указ. положение P0 и P1 относит. взаимногоисключения */
int turn;        /* Если 0 - может работать P0, если 1 - P1 */
```

```
void P0() {
    /* ОСТАЛЬНОЙ КОД */
    flag[0]=true;    /* P0 хочет войти в критический раздел */
    turn=1;          /* Уступка очереди */
    while(flag[1] && turn==1); /* и АКТИВНОЕ ОЖИДАНИЕ */
    /* КРИТИЧЕСКИЙ РАЗДЕЛ */
    flag[0]=false;   /* P0 вышел из критического раздела */
    /* ОСТАЛЬНОЙ КОД */
}
```

```
void P1() {
    /* ОСТАЛЬНОЙ КОД */
    flag[1]=true;    /* P1 хочет войти в критический раздел */
    turn=0;          /* Уступка очереди */
    while(flag[0] && turn==0); /* и АКТИВНОЕ ОЖИДАНИЕ */
    /* КРИТИЧЕСКИЙ РАЗДЕЛ */
    flag[1]=false;   /* P1 вышел из критического раздела */
    /* ОСТАЛЬНОЙ КОД */
}
```

```
void main() {
    flag[0]=false;
    flag[1]=false;
    parbegin(P0,P1); /* Гипотетическая процедура для запуска P0 и P1 */
}
```



Отключение прерываний.

Как видим, чисто программное взаимное исключение довольно нетривиальная задача, требующая к тому же использования нескольких глобальных переменных. Возможны ли более простые решения?


В однопроцессорной системе для того, чтобы гарантировать взаимное исключение достаточно защитить П от прерывания. При этом структура процедуры выглядит след. образом:

Листинг 4

```
/* БЕЗОПАСНЫЙ КОД */;  
Запретить аппаратные прерывания;  
/* КРИТИЧЕСКИЙ РАЗДЕЛ */;  
Разрешить прерывания;  
/* БЕЗОПАСНЫЙ КОД */;
```

Недостатки такого подхода: невозможность эффективной работы диспетчера и невозможность применения такого подхода в многопроцессорной архитектуре (**подумайте, почему?**).

Использование специальных машинных команд.

Как вы уже наверное поняли, неработоспособность программы из лист. 2 связана с возможностью переключения процессов в точке  после проверки состояния флага `flag` но до установки его значения. Устранить проблему можно, например, объединением проверки состояния флага и установки его значения в одной машинной операции. В этом случае обеспечивается корректный вход в критическую секцию любого П, выполняющегося на одно- или многопроцессорной системе, т. к. ячейка памяти, хранящая флаг, на время машинного цикла оказывается защищенной от доступа к ней со стороны остальных процессоров.

Определим операцию проверки и установки значения, как приведено в [1]:

Листинг 5

```
boolean testset(int i){  
    if(i==0){  
        i=1;  
        return true;  
    }  
    else  
        return false;  
}
```

Это 1 машинная команда !



В деталях реализация команды `testset` на разных компьютерах отличается. Так, процессоры семейства Pentium имеют команду `CMPSXCHG reg/mem, reg1` (сравнить и заменить) использующую три операнда: операнд-источник в регистре `reg1`, операнд-приемник в регистре или в памяти `reg/mem` и аккумулятор (т.е. регистр `AL`, `AX` или `EAX` в зависимости от размерности операндов). Если значения в приемнике и в аккумуляторе равны, значение `reg/mem` заменяется на значение `reg1`. В противном случае значение `reg/mem` загружается в аккумулятор. Флаги отражают результат, который получается при вычитании операнда назначения из аккумулятора. Флаг `ZF` устанавливается, если значения `reg/mem` и аккумулятора равны, в противном случае флаг очищается.

А вот реализация аппаратной поддержки взаимных исключений с использованием инструкции проверки и установки:

Листинг 6

```
const int n=/* кол-ство процессов */;
int bolt;

void P(int i){
    /* Остальной код */;
    while(!testset(bolt));    /* АКТИВНОЕ ОЖИДАНИЕ */
    /* Критический раздел */;
    bolt=0;
    /* Остальной код */;
}

void main(){
    bolt=0;
    parbegin(P(1),P(2),...,P(n));
}
```

Преимущества подхода с использованием аппаратной поддержки:

- 👍 Применим к любому количеству ПП как в одно-, так и в многопроцессорных системах.
- 👍 Прост, а потому легко проверяем.
- 👍 Может использоваться для поддержки множества критических разделов (каждый из них определен своей собственной переменной).

Недостатки:

- 👎 Используется пережидание занятости. Т.е. в ожидании входа в критический раздел ПП продолжает потреблять процессорное время.
- 👎 Возможно голодание. Если ПП покидает критический раздел, а очереди на вход ожидают несколько других, то выбор произволен.
- 👎 Возможна взаимоблокировка. Пусть P1 выполняет testset и входит в критический раздел, а затем P1 прерывается P2 с более высоким приоритетом. Если P2 попытается обратиться к тому же ресурсу, ему будет отказано и он войдет в цикл ожидания. Однако и P1 не может продолжить выполнения, т.к. имеется активный ПП с более высоким приоритетом (т. н. **проблема инверсии приоритетов**).

Использование спин-блокировок в Windows NT

Спин-блокировка - простейший механизм синхронизации потоков, используемый в ядре ОС семейства Windows NT. Одним из важных достоинств этого механизма, как отмечалось выше, есть простота его реализации для многопроцессорных систем (Windows NT поддерживает симметричную многопроцессорность). Рассмотрим как разработчики постарались ослабить влияние известных недостатков данного механизма.

В NT имеется два вида спин-блокировок:

- Обычные спин-блокировки, особым случаем которых являются спин-блокировки отмены запроса ввода/вывода.
- Спин-блокировки синхронизации прерываний.

И те и другие являются объектами ядра ОС. Они предназначены в основном для защиты данных, доступ к которым производится на повышенных уровнях IRQL (Interrupt ReQuest Level - уровни запросов прерываний).

При этом крайне важно разрешить возможную проблему инверсии приоритетов. Для этого нужен механизм, не позволяющий коду с некоторым уровнем IRQL прерывать код с более низким уровнем IRQL в тот момент, когда последний владеет спин-блокировкой. Таким механизмом в NT является повышение текущего уровня IRQL потока в момент захвата им спин-блокировки до некоторого уровня, ассоциированного со спин-блокировкой, и восстановление прежнего IRQL потока после ее освобождения. Код, работающий на повышенном уровне IRQL, не имеет права обращаться к ресурсу, защищенному спин-блокировкой у которой уровень IRQL ниже, чем у вызывающего кода.

С обычными спин-блокировками связан IRQL DISPATCH_LEVEL [4], то есть:

- Все попытки их захвата должны производиться на уровне IRQL, меньшим или равным DISPATCH_LEVEL.
- В случае захвата спин-блокировки текущий уровень IRQL поднимается до DISPATCH_LEVEL.

Со спин-блокировками синхронизации прерываний связан один из уровней DEVICE_LEVEL.

Некоторые функции работы с обычными спин-блокировками:

`VOID KeInitializeSpinLock(PKSPIN_LOCK SpinLock);` - инициализирует объект ядра KSPIN_LOCK.

`VOID KeAcquireSpinLock(PKSPIN_LOCK SpinLock, PKIRQL OldIrql);` - функция захвата спин-блокировки. Возвращает управление только после успешного захвата блокировки. Параметр `oldIrql` возвращает уровень IRQL, который был до захвата блокировки.

`VOID KeReleaseSpinLock(PKSPIN_LOCK SpinLock, PKIRQL NewIrql);` - функция освобождает спин-блокировку и устанавливает уровень IRQL в значение `NewIrql` (это должно быть то же значение, что и в `oldIrql`).

Спин-блокировки ядра Linux

Спин-блокировки благодаря простоте, способности работать в многопроцессорных системах и возможности использования в программных компонентах которые не могут "засыпать", таких как драйверы, широко используются в ядре ОС Linux. Приведем функции ядра Linux для работы со спин-блокировками, аналогичные по назначению рассмотренным выше:

`void spin_lock_init(spinlock_t *lock);` - инициализирует спин-блокировку. Здесь `spinlock_t` - абстрактный тип для объявления блокировки.

`void spin_lock(spinlock_t *lock);` - функция захвата спин-блокировки. Возвращает управление только после успешного захвата блокировки.
`void spin_unlock(spinlock_t *lock);` - функция освобождает спин-блокировку.

Кстати, первым ядром Linux, поддерживающим многопроцессорные системы, было 2.0 имевшее одну спин-блокировку. Эта **большая блокировка ядра** `lock_kernel` превращала все монолитное ядро в единую критическую секцию так, что в любой момент только один процессор мог выполнять код ядра. Современное ядро Linux вместо этого содержит множество "точечных" блокировок, защищающих отдельные ресурсы [J. Corbet, A. Rubini, G. Kroah-Hartman "Linux Device Drivers". 2005].

Семафоры.

Дейкстра [Dijkstra E. "Cooperating Sequential Processes". 1965] предложил механизм сотрудничества двух или более П посредством простых сигналов, так что в определенном месте П может приостановить работу, пока не дождется соответствующего сигнала.

Для сигнализации используются специальные переменные, называемые **семафорами**. Для передачи сигнала через семафор `s` П выполняет примитив `signal(s)`, а для получения сигнала - примитив `wait(s)`. В последнем случае П **приостанавливается** ("засыпает") до получения сигнала. Как правило имеется также примитив `init` для присвоения семафору начального значения. Семафор рассматривают как целое, над которым определено 3 операции:

1. Семафор может быть инициализирован неотрицательным значением.
2. Операция `wait` уменьшает значение семафора. Если значение становится отрицательным, П вызвавший `wait`, блокируется.
3. Операция `signal` увеличивает значение семафора. Если это значение не положительно, то заблокированный операцией `wait` П деблокируется.

Не имеется никаких иных способов получения информации о значении семафора или изменения его состояния. Предполагается, что примитивы `wait` и `signal` **атомарны** (не могут быть прерваны в процессе выполнения). Механизм семафоров, при реализации в стиле ООП, помимо прочего, служит для скрытия переменных блокировки так, что доступ к ним возможен только через вызовы методов объекта «семафор».

Более формальное определение примитивов семафоров (взято из [1], с другим определением можно познакомиться в [2]):

Листинг 8

```
struct semaphore{
    int count;           /* переменная блокировки */
    queueType queue; /* очередь П, ожидающих семафор */
}
void wait(semaphore s){
    s.count--;
    if (s.count<0){
        /* Поместить процесс в s.queue */;
        /* Заблокировать процесс */;
    }
}
void signal(semaphore s){
    s.count++;
    if (s.count<=0){
        /* Удалить процесс из s.queue */;
        /* Поместить процесс в список активных */;
    }
}
```

Такие **семафоры-счетчики** удобно использовать для разделения между П множества однородных ресурсов. Для этого инициализируем переменную count значением исходного количества ресурсов в пуле.

Для защиты единственного ресурса можно использовать семафор более простого вида. **Бинарный семафор** может принимать только значения 0 (блокированное) и 1 (неблокированное): лист. 9, взят из [1]:

Листинг 9

```
struct binary_semaphore{
    enum{zero,one} value;
    queueType queue;
}
void waitB(binary_semaphore s){
    if (s.value==one)
        s.value=zero;
    else {
        /* Поместить процесс в s.queue */;
        /* Заблокировать процесс */;
    }
}
void signalB(binary_semaphore s){
    /* функция is_empty() возвращает значение >0 если очередь пуста */
    if (is_empty(s.queue))
        s.value=one;
    else{
        /* Удалить процесс из s.queue */;
        /* Поместить процесс в список активных */;
    }
}
```

Упрощенная версия бинарных семафоров наз. **мьютексами** (mutex, сокращ. от mutual exclusion). Для них обычно исп. обозначения примитивов mutex_lock и mutex_unlock вместо wait и signal соответственно.

Неблокированному состоянию мьютекса соответствует значение 0, а все остальные - блокированному. Рассмотрим пример реализации мьютекса для процессоров фирмы Intel (не использующий всех возможностей команды `SMPLXCHG`) из листинга 10. Здесь используется процедура уступки процессора другому потоку `thread_yield`. Поток, вызвавший `thread_yield`, приостанавливает свою работу и передает управление планировщику потоков который, в свою очередь, активирует другой поток данного П. Поскольку планировщик потоков работает как и потоки в пользовательском адресном пространстве, выполнение `mutex_lock` и `mutex_unlock` не требует переключения в режим ядра, а значит происходит быстро!

Листинг 10

```
; Данные программы
DATA SEGMENT
    mutex DW 0 ; Объявление и инициализация переменной
DATA ENDS

; Код программы
CODE SEGMENT
; .....
mutex_lock: ; Реализация примитива захвата мьютекса для i486+
    MOV ax,0 ; будет проводится проверка переменной на 0
    MOV bx,1 ; и замена на 1
    CMPLXCHG mutex,bx; это реализация testset для i486+ (см. выше)
    ; если ax == mutex, то bx -> mutex и установить флаг zf
    JE ok ; переход если было mutex == 0 (по флагу zf)
    CALL thread_yield ; иначе уступить процессор другому потоку
    JMP mutex_lock ; и повторить попытку захвата мьютекса
ok: RET

mutex_unlock: ; Реализация примитива освобождения мьютекса
    MOV mutex,0 ; выполняется обычное обнуление переменной
    RET
; .....
CODE ENDS
```

Для хранения П и/или потоков, ожидающих как обычные семафоры, так и мьютексы, используется очередь. Наиболее корректно для очереди использовать принцип FIFO (т. н. **сильный семафор** - strong semaphore). При этом первым из очереди выбирается П, который был заблокирован дольше других. Если порядок извлечения из очереди не определен - семафор наз. **слабым** (weak semaphore). Обычно ОС используют сильные семафоры, т.к. они удобнее и гарантируют отсутствие голодания.

Реализация семафоров.

Выше мы рассмотрели пример реализации мьютекса с использованием команды `SMPLXCHG` - конкретной реализации инструкции проверки и установки `testset`. Приведем более общие алгоритмы

реализации семафора, определенного в лист. 8.

Условие корректности работы семафоров заключается в атомарности операций wait и signal. Оптимально реализовать их в аппаратном или микропрограммном виде. Иначе применяют различные программные решения. Можно Деккера или Петерсона (большие накладные расходы) или использовать одну из схем поддержки взаимного исключения на аппаратном уровне. В однопроцессорной системе можно воспользоваться приемом запрета прерываний.

Листинг 11

```
/* Используя инструкцию проверки и установки */
wait(s){
    while(testset(s.flag)); /* Ждем */
    s.count--;
    if(s.count<0){
        /* Поместить П в s.queue заблокировать П и установить s.flag=0 */
    }
    else
        s.flag=0;
}
signal(s){
    while(testset(s.flag)); /* Ждем */
    s.count++;
    if(s.count<=0){
        /* Удалить П из s.queue поместить П в список активных П */
    }
    s.flag=0;
}
```

Листинг 12

```
/* Используя запрет прерываний */
wait(s){
    /* Запретить прерывания */;
    s.count--;
    if(s.count<0){
        /* Поместить П в s.queue заблокировать П */
    }
    else
        /* Разрешить прерывания */;
}
signal(s){
    /* Запретить прерывания */;
    s.count++;
    if(s.count<=0){
        /* Удалить П из s.queue поместить П в список активных П */
    }
    /* Разрешить прерывания */;
}
```

В первом случае неизбежно пережидание занятости, но т.к. операции wait и signal относительно небольшие - время ожидания минимально. Во втором случае запрет прерываний также непродолжителен по той же причине.

Семафоры в языках программирования и ОС.

Семафоры - механизм взаимных исключений, который может предоставляться языком программирования или ОС. По видимому первым языком программирования, получившим семафоры, стал Algol-68. Прimitives, названные в данном пособии `wait` и `signal`, в Algol-68 получили имена `down` и `up` соответственно.

Другой классический язык программирования, поддерживающий синхронизацию П посредством семафоров - Модула-2. Рекомендуемый автором языка в качестве стандартного модуль определений `Processes` [3] приведен на лист. 13:

Листинг 13

```
DEFINITION MODULE Processes;
  TYPE SIGNAL;

  PROCEDURE StartProcess(P:PROC; n:CARDINAL); (* Начать параллельный П,
      задаваемый программой P с рабочей областью размером n *)
  PROCEDURE SEND (VAR s:SIGNAL); (* Возобновляет один из П, ждущих s
  *)
  PROCEDURE WAIT (VAR s:SIGNAL); (* Ждать пока не пришьют сигнал s *)
  PROCEDURE Awaited (VAR s:SIGNAL): BOOLEAN; (* Если TRUE - хотя бы
      один П ждет s *)
  PROCEDURE Init (VAR s:SIGNAL); (* Обязательная инициализация *)
END Processes.
```

Отметим наличие процедуры обязательной инициализации семафора `Init` и процедуру проверки очереди П, ожидающих семафора `Awaited` (сравни с `is_empty()` из лист. 9).

Что касается современных систем программирования на языках C/C++, то они, как правило, имеют в своем составе библиотеки поддержки средств синхронизации процессов опирающиеся на возможности, предоставляемые ОС. Так например, библиотека MFC фирмы Microsoft содержит классы `CSemaphore` и `CMutex`, предоставляющие механизм семафоров для многопоточных приложений (рис. 1).

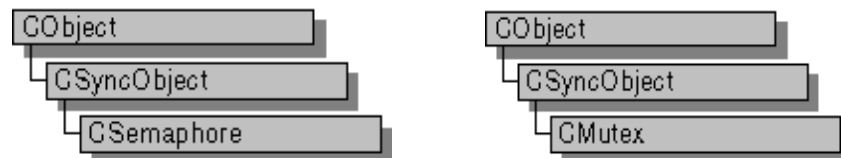


Рис. 1. Иерархия классов MFC для синхронизации потоков.

Рассмотрение поддержки взаимных исключений средствами ОС начнем из систем семейства UNIX. Системные вызовы для управления потоками в UNIX-подобных ОС стандартизованы в части стандарта POSIX (P1003.1c). Стандартом не оговаривается способ реализации вызовов, так что они могут быть как настоящими вызовами сервисов ядра системы, так и вызовами библиотечных функций, работающих в пространстве пользователя.

Таблица 2. Некоторые вызовы POSIX для работы с семафорами

Вызов	Описание
<code>pthread_mutex_init</code>	Создает новый мьютекс
<code>pthread_mutex_destroy</code>	Уничтожает мьютекс
<code>pthread_mutex_lock</code>	Блокировка мьютекса (см. <code>wait</code>)
<code>pthread_mutex_unlock</code>	Разблокирование мьютекса (см. <code>signal</code>)
<code>sem_init</code>	Инициализирует семафор
<code>sem_wait</code>	Увеличивает значение счетчика (см. <code>wait</code>)
<code>sem_post</code>	Увеличивает значение счетчика (см. <code>signal</code>)
<code>sem_getvalue</code>	Считывает тек. значение счетчика семафора

Заметьте, что семафоры в стандарте POSIX являются динамическими объектами. Кроме мьютексов и семафоров-счетчиков стандарт предлагает механизм **переменных состояния** для долговременной синхронизации. Подробнее об использовании переменных состояния см. в [2].

Что касается конкретных реализаций UNIX систем, то все они предоставляют богатый набор механизмов синхронизации П. Так например, UNIX System V имеет семафоры-счетчики состоящие из следующих элементов:

- Текущего значения семафора.
- Идентификатора последнего П, работавшего с семафором.
- Количества П, ожидающих, пока значение семафора не увеличится.
- Количества П, ожидающих, пока значение семафора не станет равным 0.

Семафоры при создании объединяются в множества, содержащие один или несколько семафоров. Благодаря этому имеется возможность одновременного выполнения ядром ОС групповых операций с множеством семафоров.

ОС Solaris поддерживает четыре примитива синхронизации потоков:

- Блокировки взаимоисключений (мьютексы).
- Семафоры-счетчики.
- Блокировки читатели-писатель (несколько читателей, один писатель).
- Переменные условий

Для работы с мьютексами используются системные вызовы `mutex_enter()`, `mutex_exit()`, `mutex_tryenter()`. Первые два обеспечивают захват и освобождение мьютекса соответственно, а третий позволяет определить текущее состояние мьютекса и, таким образом, использовать технологию пережидания занятости для избежания блокирования всего П из-за блокирования одного из потоков.

Соответствующие вызовы для семафоров-счетчиков имеют имена `sema_p()`, `sema_v()`, `sema_tryv()`.

Об остальных двух типах примитивов синхронизации Solaris позже.

ОС Linux в основном наследовала механизмы синхронизации UNIX System V, в том числе и групповые операции с множеством семафоров (на лист. 14 приведены объявления групповых операций из файла <sys/sem.h>):

Листинг 14

```
/* Structure used for argument to `semop' to describe operations. */
struct sembuf {
    unsigned short int sem_num; /* semaphore number */
    short int sem_op;           /* semaphore operation */
    short int sem_flg;          /* operation flag */
};
/* Semaphore control operation. */
extern int semctl (int __semid, int __semnum, int __cmd, ...);
/* Get semaphore. */
extern int semget (key_t __key, int __nsems, int __semflg);
/* Operate on semaphore. */
extern int semop (int __semid, struct sembuf *__sops, size_t __nsops);
```

Кроме того, Linux поддерживает (на уровне системной библиотеки **glibc**) средства синхронизации стандарта POSIX 1003.1b (с некоторыми ограничениями). Более подробно с соответствующими средствами ОС Linux можно ознакомиться в электронном руководстве Map pages части 2 и 3.

В ОС семейства Windows NT набор механизмов синхронизации, рассчитанных на применение при пониженных уровнях IRQ, объединяется понятием **диспетчерского объекта**. Общим свойством любого диспетчерского объекта является то, что в каждый момент времени такой объект находится в одном из двух состояний - **сигнальном** или **несигнальном**, а поток, ожидающий захвата объекта, блокирован. В этом плане единственным отличием одного диспетчерского объекта от другого является условие, по которому меняется состояние объекта (переход в сигнальное или несигнальное состояние). Диспетчерские объекты и правила изменения их состояния перечислены в табл. 3 из [4]:

Таблица 3.

Диспетчерские объекты Windows NT

Тип объекта	Переход в сигнальное состояние	Результат для ожидающих потоков
Мьютекс	Освобождение мьютекса	Освобождается один из ожидающих потоков
Семафор	Счетчик захватов становится ненулевым	Освобождается некоторое количество ожидающих потоков
Событие синхронизации	Установка события в сигнальное состояние	Освобождается один из ожидающих потоков
Событие оповещения	Установка события в сигнальное состояние	Освобождается все ожидающие потоки
Таймер синхронизации	Наступило время или истек интервал	Освобождается один из ожидающих потоков
Таймер оповещения	Наступило время или истек интервал	Освобождается все ожидающие потоки
Процесс	Завершился последний поток процесса	Освобождается все ожидающие потоки
Поток	Завершился поток	Освобождается все ожидающие потоки
Файл	Завершена операция ввода/вывода	Освобождается все ожидающие потоки

Рассмотрим подробнее два типа диспетчерских объектов Windows NT - мьютексы и семафоры. Об остальных поговорим позже.

Мьютексы в Windows NT имеютс^я двух типов - **мьютексы ядра** (или просто мьютексы) и **быстрые мьютексы**, являющиеся объектами исполнительной системы и не являющимися диспетчерскими объектами.

Для мьютексов ядра выполняются следующее:

- Захват мьютекса происходит в контексте конкретного потока. Этот поток является владельцем мьютекса и может захватывать его рекурсивно. **Кстати, машинная команда `SMRХSNG` (см. выше) в случае неуспеха захвата мьютекса возвращает идентификатор его текущего владельца.** Драйвер, захвативший мьютекс в контексте потока, обязан освободить его в том же контексте, иначе система будет разрушена.
- Для мьютексов предусмотрен механизм исключения взаимоблокировок. При инициализации мьютекса функцией `KeInitializeMutex()` указывается уровень (level) мьютекса. Если потоку требуется захватить несколько мьютексов одновременно, он должен сделать это в порядке возрастания уровня level.

Основные функции работы с мьютексами ядра:

`VOID KeInitializeMutex(PKMUTEX Mutex, ULONG Level)` - инициализирует мьютекс. Память под мьютекс уже должна быть выделена. После инициализации мьютекс в сигнальном состоянии.

`LONG KeReleaseMutex(PKMUTEX Mutex, BOOLEAN Wait)` - освобождает мьютекс (переводит из несигнального состояния в сигнальное). Если `wait` равен `TRUE`, то следующим вызовом должен быть вызов ожидания (захвата) мьютекса. Тогда гарантируется, что пара этих вызовов будет выполнена как одна операция без переключения контекста потока (рекурсивный захват).

`KeWaitForSingleObject()` - ожидание (захват объекта) момента перехода диспетчерского объекта (не обязательно мьютекса!), указанного параметром, в сигнальное состояние. Задается также интервал времени, в течении которого необходимо ожидать события.

`KeWaitForMultipleObject()` - в отличие от предыдущей может ожидать перехода в сигнальное состояние сразу всех указанных в ней объектов, либо любого из них.

`LONG KeReadStateMutex(PKMUTEX Mutex)` - возвращает состояние мьютекса.

Для работы с мьютексами на пользовательском уровне используются функции `CreateMutex()`, `OpenMutex()`, `ReleaseMutex()`, `WaitforSingleObject()`, `WaitForMultipleObjects()`, `CloseHandle()` и другие.

Быстрые мьютексы не являются диспетчерскими объектами и не могут быть рекурсивно захвачены. Для работы с ними имеется отдельный набор функций - `ExInitializeFastMutex()`, `ExAcquireFastMutex()`,

ExReleaseFastMutex() и другие. Аналогов быстрых мьютексов на пользовательском уровне нет, они могут использоваться только в режиме ядра.

Семафоры ядра Windows NT инициализируются функцией `VOID KeInitializeSemaphore(PKSEMAPHORE Semaphore, LONG Count, LONG Limit)`, где `Count` - начальное значение семафора (0 - несигнальное состояние, >0 - сигнальное), `Limit` - максимальное значение `Count` (максимальное число свободных ресурсов). Функция `KeReleaseSemaphore()` увеличивает счетчик семафора на указанное параметром значение (освобождает указанное число ресурсов). При вызове функции ожидания `WaitforSingleObject()` или `WaitForMultipleObjects()` счетчик уменьшается на 1 для каждого разблокированного потока. По достижению 0 семафор переходит в несигнальное состояние. Занять семафор может один поток, а освободить - другой. Поэтому при разработке драйверов использовать семафоры необходимо с осторожностью (почему?).

Для работы с семафорами на пользовательском уровне используются (подобно мьютексам) функции `CreateSemaphore()`, `OpenSemaphore()`, `ReleaseSemaphore()` и другие.

Использование семафоров для взаимных исключений.

Пример использования семафоров для реализации взаимных исключений в листинге 15 повторяет решение задачи взаимоисключений из листинга 7. Сравните их.

Листинг 15

```
const int n=/* кол-ство процессов */;
semaphore s;

void P(int i){
    while(true){
        wait(s);
        /* Критический раздел */;
        signal(s);
        /* Остальной код */;
    }
}

void main(){
    init(s,1);          /* инициализирует s.count значением 1 */
    parbegin(P(1),P(2),...,P(n));
}
```

Задача производителя/потребителя.

Одной из фундаментальных задач параллельных вычислений является задача производителя/потребителя. Предполагается, что имеется один или несколько производителей, генерирующих данные (элементы) некоторого типа (расходуемый ресурс) и помещающих их в буфер (повторно используемый ресурс) и единственный потребитель, извлекающий из буфера элементы по одному и использующий их.

Требуется защитить программную систему от перекрытия операций с буфером, т. е. обеспечить одновременный доступ к буферу только для одного процесса.

Можно предложить множество решений с использованием тех или иных средств синхронизации. Одно из них приведено в Лист. 16:

Листинг 16

```
#include <pthread.h> /* Объявления POSIX-функций для потоков */
#include <semaphore.h> /* Объявления POSIX-функций для семафоров */

const int SizeOfBuffer = 64; /* Такого размера будет буфер */
const int NIL = 0; /* Просто ноль */
pthread_t ProducerID, ConsumerID; /* Идентификаторы потоков */
sem_t s; /* Этот семафор защищает буфер */
sem_t occupied; /* Кол-ство элементов в буфере */
sem_t available; /* Свободное место в буфере */
char chin, chout;
char Buffer[SizeOfBuffer]; /* Вот он буфер для символов */

void Produce() { /* Здесь реализация производства элемента chin */ }
void Consume() { /* Здесь реализация использования элемента chout */ }

void Producer() {
    static int NextFree = 0;
    while(true) {
        Produce(); /* Производство элемента */
        sem_wait(&available); /* Ждем свободного места в буфере */
        sem_wait(&s); /* и пытаемся получить доступ к нему */
        /* Так помещают элемент в кольцевой буфер: */
        Buffer[NextFree] = chin;
        NextFree = (NextFree + 1) % SizeOfBuffer;
        sem_post(&s); /* Конец критического раздела */
        sem_post(&occupied); /* Плюс один элемент */
    }
}

void Consumer() {
    static int NextChar = 0;
    while(true) {
        sem_wait(&occupied); /* Ждем хоть одного элемента в буфере */
        sem_wait(&s); /* и пытаемся получить доступ к нему */
        /* А так выбирают из кольцевого буфера: */
        chout = Buffer[NextChar];
        NextChar = (NextChar + 1) % SizeOfBuffer;
        sem_post(&s); /* Конец критического раздела */
        sem_post(&available); /* Плюс одно свободное место */
        Consume(); /* Использование элемента */
    }
}

void main() {
    /* ИНИЦИАЛИЗИРУЕМ СЕМАФОРЫ */
    sem_init(&s, NIL, 1); /* Буфер доступен */
    sem_init(&occupied, NIL, 0); /* Вначале в буфере нет ничего, т.е. */
    sem_init(&available, NIL, SizeOfBuffer); /* аж вот столько места */
    /* И СТАРТУЕМ ПОТОКИ */
    pthread_create(&ProducerID, NULL, Producer, NULL);
    pthread_create(&ConsumerID, NULL, Consumer, NULL);
} /* Для тех, кто забыл: NULL - это нулевой указатель */
```

Здесь для взаимоисключений при работе с кольцевым буфером одного производителя и потребителя использованы семафоры-счетчики. Использована реализация функций работы с семафорами и потоками в стандарте POSIX. После некоторой доработки программу можно откомпилировать и выполнить в ОС Linux. Заметьте, что из трех семафоров только один используется "по прямому назначению". Подумайте, как можно обойтись без двух других, не вводя дополнительных переменных. Чем такое решение может быть лучше? А чем хуже? Что будет если в тексте потребителя поменять местами вызовы `sem_post`? А если поменять местами вызовы `sem_wait`?

Барьеры.

Этот механизм предназначен для синхронизации множества ПТ или потоков, которые время от времени должны все находиться в некотором вполне определенном состоянии.

Типовые задачи, требующие такого типа синхронизацию ПТ - моделирование динамики пучков заряженных частиц, плазмы или галактик. При этом рассчитывается перемещение тысяч - сотен тысяч частиц за небольшой отрезок времени, затем по их новому положению рассчитываются действующие между ними силы и этот цикл многократно повторяется. Расчеты перемещения частиц можно выполнять параллельно. Однако расчет действующих сил можно начинать только после того, как была перемещена последняя частица.

Такую синхронизацию реализуют, размещая в конце каждой фазы выполнения программы **барьер**. Когда ПТ доходит до барьера, он блокируется (вызывая функцию `barrier`) до тех пор, пока все остальные ПТ не дойдут до барьера. Затем блокировка снимается одновременно со всех ПТ. Как правило, барьеры реализованы в составе программного обеспечения кластеров. Так, стандарт программного интерфейса библиотеки MPI (Message Passing Interface - интерфейс передачи сообщений) определяет функцию `MPI_BARRIER()`, обладающую описанными свойствами.

Мониторы.

Семафоры предоставляют мощный и гибкий механизм реализации безопасного межпроцессного взаимодействия. Однако гибкость всегда имеет обратную сторону - программы, создаваемые с использованием таких средств, подвержены ошибкам, причиной которых является их неизбежно повышенный уровень сложности.

В качестве иллюстрации можно опять обратиться к программе из листинга 16. Вы уже видели, что случайное изменение порядка вызовов примитивов синхронизации способно сделать программу неработоспособной, притом это не слишком очевидно. А ведь это всего лишь коротенький учебный пример. Ошибки, связанные с неправильным

использованием семафоров, могут приводить к непредсказуемым и невоспроизводимым критическим состояниям вычислительной системы.

Отсюда очевидна необходимость, в первую очередь для целей прикладного программирования, иметь средство синхронизации более высокого уровня, которое могло бы обеспечить повышение надежности параллельных программ. Таким средством стали **мониторы**, предложенные Хоаром (Hoare) и Бринч Хансеном (Brinch Hansen) в 1974 г. Основная идея состоит в том, чтобы «спрятать» защищаемый ресурс внутри объекта, обеспечивающего взаимоисключения.

Мониторы с сигналами.

Монитор (версия Хоара - Хансена) представляет собой программный модуль, состоящий из инициализирующей последовательности, одной или нескольких процедур и локальных данных. Основные характеристики:

1. Локальные переменные монитора доступны только его процедурам; внешние процедуры доступа к локальным данным монитора не имеют.
2. Процесс входит в монитор путем вызова одной из его процедур.
3. В мониторе в определенный момент времени может выполняться только один процесс; любой другой процесс, вызвавший монитор, будет приостановлен в ожидании доступности монитора.

Очевидно (требования 1,2), объектно ориентированные ОС или языки программирования могут легко реализовать монитор как объект со специальными характеристиками. Соблюдение требования 3 позволяет монитору обеспечить взаимоисключения. Если данные в мониторе представляют некий ресурс, то монитор обеспечивает взаимоисключение при обращении к ресурсу. Для применения в параллельных вычислениях мониторы должны обеспечивать синхронизацию. Пусть П находясь в мониторе должен быть приостановлен до выполнения некоторого условия. При этом требуется механизм, который не только приостанавливает П, но и освобождает монитор, позволяя войти в него другому П. Позже, по выполнению условия, и когда монитор станет доступным, приостановленный П сможет продолжить свою работу.

Монитор поддерживает синхронизацию при помощи **переменных условия**, располагающихся и доступных только в мониторе. Работать с этими переменными могут две ф-ции.

- `cwait(c)`: приостанавливает выполнение вызывающего П по условию `c`. Монитор при этом доступен для использования другим П.
- `csignal(c)`: возобновляет выполнение некоторого П, приостановленного вызовом `cwait` с тем же условием. Если имеется несколько таких П, выбирается один из них; если таких П нет, ф-ция не делает ничего.

Как видим, операции `cwait/csignal` монитора отличаются от соотв. операций семафора. Если П в мониторе передает сигнал, но при этом нет ни одного ожидающего его П, то сигнал просто теряется. Это значит, что переменные условия, в отличие от семафоров, не являются счетчиками.

В качестве иллюстрации использования монитора повторим решение задачи производитель/потребитель с ограниченным буфером:

Листинг 17

```
monitor BoundedBuffer {
    char Buffer[N];          /* Буфер на N элементов */
    int NextIn, NextOut;     /* Текущие указатели буфера */
    int Count;              /* Тек. кол-ство элементов в буфере */
    int NotFull, NotEmpty;  /* Синхронизация (переменные условия)
*/
    void Append(char x) {    /* Точка входа в монитор для производителя
*/
        if (Count==N)       /* Если буфер заполнен - */
            cwait(NotFull); /* блокируемся (потребитель может работать)*/
        Buffer[NextIn] = x;
        NextIn = (NextIn+1)%N;
        Count++;            /* Добавили элемент в буфер */
        csignal(NotEmpty);  /* Возобновление работы потребителя */
    }

    void Take(char x){       /* Точка входа в монитор для потребителя */
        if (Count==0)       /* Если буфер пуст - */
            cwait(NotEmpty); /*блокируемся (производитель может работать)*/
        x = Buffer(NextOut);
        NextOut = (NextIn+1)%N;
        Count--;           /* Удалили элемент из буфера */
        csignal(NotFull);  /* Возобновили работу производителей */
    }

    /* ТЕЛО МОНИТОРА */
    NextIn = 0;             /* Вначале буфер пуст */
    NextOut = 0;
    Count = 0;
}

void Producer(){           /* Производитель */
    char x;
    while(true){
        Produce(x);        /* Генерация данных */
        Append(x);         /* Войти в монитор, добавить элемент и выйти */
    }
}

void Consumer(){           /* Потребитель */
    char x;
    while(true){
        Take(x);           /* Войти в монитор, взять элемент и выйти */
        Consume(x);        /* Использование данных */
    }
}

void main(){
    parbegin(producer, consumer);
}
```

Мониторы с оповещением и широковещанием.

Хоаровское определение мониторов требует, чтобы в случае, если очередь ожидания выполнения условия не пуста, при выполнении каким-либо П операции `csignal` для этого условия был немедленно запущен П из указанной очереди. Т.о., выполнивший `csignal` П должен либо немедленно выйти из монитора, либо быть приостановленным.

У такого подхода 2 недостатка:

1. Если выполнивший `csignal` П не завершил свое пребывание в мониторе, то требуется 2 дополнительных переключения П: для приостановки данного П и для возобновления его, когда монитор станет доступен.
2. Планировщик П, связанный с сигналом, должен быть идеально надежен. При выполнении `csignal` П из соответствующей очереди должен быть немедленно активизирован до того, как в монитор войдет другой П и условие активации при этом может измениться. В результате возможна взаимная блокировка всех П, связанных с данным условием.

Lampson и Redell разработали другое определение монитора для языка Mesa (такая же структура монитора использована и в языке Modula-3). Примитив `csignal` заменен примитивом `cnotify(x)`, который интерпретируется след. образом. Если П, выполняющийся в мониторе, вызывает `cnotify(x)`, об этом оповещается очередь условия `x`, но выполнение П продолжается. Результат оповещения состоит в том, что П в начале очереди условия возобновит свою работу в ближайшем будущем, когда монитор окажется свободным. Однако, поскольку нет гарантии, что другой П не войдет в монитор раньше, при возобновлении работы П должен проверить, выполнено ли условие.

Тогда процедуры монитора `BoundedBuffer` из Лист.17 будут иметь следующий вид [1]:

Листинг 18

```
void Append(char x) {
    while (Count==N)           /* Если буфер заполнен - */
        cwait(NotFull); /* заблокировались и затем проверим еще раз */
    Buffer[NextIn] = x;
    NextIn = (NextIn+1)%N;
    Count++;                   /* Добавим элемент в буфер */
    cnotify(NotEmpty);         /* Уведомляем потребителя */
}
void Take(char x){
    while (Count==0)           /* Если буфер пуст - */
        cwait(NotEmpty); /* заблокировались и затем проверим еще раз */
    x = Buffer[NextOut];
    NextOut = (NextOut+1)%N;
    Count--;                   /* Удалим элемент из буфера */
    cnotify(NotFull);          /* Уведомляем производителя */
}
```

Инструкции `if` заменены циклами `while`; таким образом, будет выполняться как минимум одно лишнее вычисление переменной условия. Однако в этом случае отсутствуют лишние переключения ТП и не имеется ограничений на момент запуска ожидающего ТП.

Важно, что для монитора Лемпсона-Ределла можно ввести предельное время ожидания, связанное с примитивом `notify`. ТП, который прождал уведомления в течении предельного времени, помещается в очередь активных, независимо от того, было ли уведомление о выполнении условия. При активации ТП проверяет выполнение условия и либо продолжает работу, либо опять блокируется. Это предотвращает голодание в случае, если другие ТП сбоят перед уведомлением о выполнении условия.

Также в систему команд можно включить примитив `cbroadcast` (широковещательное сообщение), который вызывает активизацию всех ожидающих ТП. Это удобно, когда уведомляющий ТП не знает о количестве ожидающих ТП или не в состоянии определить, который ТП надо вызвать.

Дополнительным преимуществом монитора Лемпсона-Ределла перед монитором Хоара, является меньшая подверженность ошибкам. Если пришло ложное оповещение, ТП все равно проверит выполнение условия.

Мониторы являются структурным компонентом языка программирования и ответственность на организации взаимного исключения лежит на компиляторе. Последний обычно использует мьютексы и переменные условий. Часто ОС предоставляют поддержку реализации мониторов именно через эти средства (см. выше ОС Solaris). Типовым представителем языков программирования, в котором мониторы являются основным средством синхронизации потоков, является Java. Добавление в описание метода ключевого слова `synchronized` гарантирует, что если один поток начал выполнение этого метода, ни один другой поток не сможет выполнять другой синхронизированный метод из этого класса. Т.е., в языке Java у каждого объекта, имеющего синхронизованные методы, есть связанный с ним монитор. Пример решения задачи производителя и потребителя на Java можно найти в [2].

Общим недостатком семафоров и мониторов является то, что они рассчитаны на работу в одно- или многопроцессорной вычислительной системе с общей памятью. Если система состоит из нескольких процессоров, каждый из которых имеет свою собственную память, а связь между ними осуществляется через каналы ввода/вывода, ни один из рассмотренных выше механизмов реализации взаимоисключений не работоспособен. В этом случае возможно только сотрудничество ТП с использованием связи.



Использованная литература

1. **Столлингс В.** Операционные системы, 4-е изд.: - М.: Издательский дом "Вильямс", 2002. - 848 с.
2. **Таненбаум С.** Современные операционные системы, 2-е изд.: - СПб.: Питер, 2004. - 1040 с.
3. **Вирт Н.** Программирование на языке Модула-2.: - М.: Мир, 1987. - 224 с.
4. **Сорокина С. И.** и др. Программирование драйверов и систем безопасности.: - СПб.: БХВ-Петербург, М.: Изд. Молгачева С.В., 2002. - 256 с.
5. *****J. Corbet, A. Rubini, G. Kroah-Hartman** Linux Device Drivers, 3 ed. - O'Reilly, 2005. - 585 p.