**1. Постановка задачи взаимного исключения.**

Необходимо согласовать работу n>1 параллельных потоков при использовании некоторого критического ресурса таким образом, чтобы удовлетворить следующим требованиям:

1. Одновременно внутри критической секции должно находиться не более одного потока.
2. Критические секции не должны иметь приоритета по отношению друг к другу.
3. Остановка какого-либо потока вне его критической секции недолжна влиять на дальнейшую работу потоков при использовании критических ресурсов.
4. Любой поток может переходить в любое состояние вне пределов своей критической секции.
5. Решение о вхождении потоков в их критические секции при одинаковом времени поступления запросов не откладывается на неопределенный срок, а является конечным во времени.
6. Освобождение критического ресурса и выход из критической секции должны быть произведены потоком, использующим критический ресурс, за конечное время.
7. Относительные скорости выполнения потоков неизвестны и произвольны.

**2. Необходимость синхронизации.**

Критическая секция — часть программы, результат выполнения которой может непредсказуемо меняться, если в ходе ее выполнения состояние ресурсов, используемых в этой части программы, изменяется другими потоками.

Критическая секция всегда определяется по отношению к определенным критическим ресурсам, при несогласованном доступе к которым могут возникнуть нежелательные эффекты.

Ситуация, когда несколько потоков работают с разделяемым ресурсом и конечный результат зависит от относительных скоростей потоков, называется состязаниями или гонками.

Требуется обеспечить, чтобы в каждый момент времени в критической секции находился только один поток. Все остальные потоки должны блокироваться на входе в критическую секцию. Когда один поток покидает кс, другой может в нее войти.

**3. Решение задачи взаимного исключения с помощью запрещения прерываний.**

while (true){

       DisableInterrupts();

       CS();

       EnableInterrupts();

       NCS();

}

Недостатки:

1. Не работает на многопроцессорных системах.
2. Прикладные программы, как правило, не могут запрещать прерывания.
3. Внешние устройства могут не получить своевременного обслуживания.

**4. Алгоритм Петерсона.**

int ready[2]={0,0}; // Номера потоков.

int turn=0; // Номер потока, имеющего приоритет при входе в критическую секцию.

while (true) {

       ready[i]=1;

       turn=1-i;

       while ((ready[1-i]) && (turn=1-i));

       CS();

       ready[i]=0;

       NCS();

}

Является приоритетным для многопроцессорных систем.

**5. Алгоритм булочной.**

**6. Решение задачи взаимного исключения с помощью специальных команд ЦП (активное ожидание).**

**I. Test&Set.**

int Test&Set (int \*a) {

        int tmp;

       tmp =\*a;

       \*a=1;

        return tmp;

} // По указанному адресу записываем 1, а возвращаем то, что было раньше.

int lock=0; // 0 — свободен, 1 — занят.

while (true ) {

        while (Test&Set(&lock)) // Как только ресурс освободится, войдем в КС.

**(\*)**      ;

       CS();

       lock=0;

       NCS();

}

**II. Swap.**

void Swap (int \*a, int \*b) {

        int tmp = \*a;

       \*a=\*b;

       \*b=tmp;

}

int lock=0, key;

while (true) {

       key=a;

       Swap (&lock, &key);

        while (key);

       CS();

       lock=0;

       NCS();

}

**Активное ожидание** — многократная попытка занять ресурс с помощью специальной команды.



Замечания:

1. В чистом виде неэффективно. Решение — вставить вместо **(\*)** (см. выше) следующий кусок кода:

yield();

sleep(0); // Добровольно освободить текущий квант.

2. Существую ситуации, когда активное ожидание эффекитивно, а при использовании сист. вызова ушло бы очень много времени.

3. ***ВОТ ЗДЕСЬ ШТА***

**7. Семафоры, мьютексы. Решение задач Читатель-Писатель и Производитель-Потребитель с помощью семафоров.**

Семафор — целочисл. неотр. переменная, на которой определены две атомарные операции (атомарность обеспечивается реализацией).

V(S) — S++; // Неблокирующая — независимо от состояния S увеличиваем на 1 и идем дальше.

P(S) — S--; // Блокирующая — если S=0, поток будет ожидать его увеличения.

Типы семафоров:

1. Двоичные семафоры. Используются для обеспечения эксклюзивного доступа к ресурсу (решение задачи вз. искл.). Если S=1, ресурс свободен, S=0 — занят. Начальное значение — 1.

Semaphore S=1;

while (true ) {

       P(S); // Захват ресурса в эксклюзивное пользование.

       CS();

       V(S); // Освобождение.

       NCS();

}

2. Счетные семафоры — используются для контроля количества доступных ресурсов. Значение семафора — текущее количество свободных единиц ресурсов. Начальное значение — общее кол-во единиц ресурсов.

Пусть даны пять принтеров.

Semaphore S1=5, S2=1;

while (true ) {

       P(S1); // Резервирование единицы ресурсов.

       P(S2); // Получение эксклюзивного доступа к структуре, описывающей принтер.

       Выбор принтера ();

       V(S2);

       Печать();

       P(S2);

       Освобождение принтера();

       V(S2);

       V(S1);

}

3. Мьютексы — специальный тип двоичных семафоров. Объект синхронизации, который может находиться в одном из двух состояний: свободен и занят. Определены две операции:

lock() / unlock()

acquire() / release()

\_\_\_

Mutex M=unlocked;

while (true ) {

       acquire(M);

       CS();

       release(m);

       NCS();

}

Дополнительные свойства Мьютекса:

1. Запоминание владельца. Освободить мьютекс может только тот поток, который его захватил.
2. Рекурсивность. Поток, захвативший мьютекс, может повторно его захватить произвольное число раз. Для освобождения нужно соответсвующее число раз вызвать release.

**Задача «Читатели-Писатели».**

Имеются данные и два типа потоков — писатели и читатели.



1. При выполнении записи доступ другим потокам запрещен.
2. При выполнении чтения доступ писателям запрещен, читателям — разрешен.
3. Потоки выполняются параллельно.

*Решение на семафорах (CS выделены цветом):*

Semaphore Access=1, RC=1; // Двоичные.

int RCount=0; // Кол-во читателей, выполняющих чтение в данный момент.

Write() {

       P(Access);

       writeOp();

       V(Access);

}

Read() {

       P(RC);

       RCount++;

        if (RCount == 1) P(Access); // Первый Ч захватывает данные для всех Ч.

       V(RC);

       ReadOp();

       P(RC);

       RCount--;

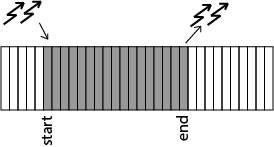
        if(RCount == 0) V(Access);

       V(RC);

}

**Задача «Производители-Потребители».**

Кольцевой буфер.



***ВОТ ЗДЕСЬ ЕЩЕ КАКОЙ-ТО ТЕКСТИК БЫЛ, НАЙТИ***

start — эл-т, который нужно потребить. Изменяется только потребителем.

end — указатель на первую пустую ячейку. Изменяется только производителем.

При пустом и полном буфере start==end, нужно как-то отличить эти две ситуации.

1. Флажок: 0 — буфер пуст, 1 — полон.
2. Пуст — start=end=-1.

Появляется переменная, изменяемая обоими потоками.

*Решение на семафорах:*

Semaphore Full=0, Empty=N, Access=1;

Produce(el) {

       P(Empty);

       P(Access);

       Добавить (el);

       V(Access);

       V(Full);

}

Consume (\*el) {

       P(Full);

       P(Access);

       \*el=Извлечь();

       V(Access);

       V(Empty);

}

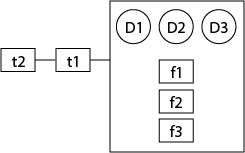
Недостаток семафоров и мьютексов в том, что их связь, защищаемая критическими ресурсами, установлена только в голове программиста. Соответственно, ОС не может автоматически контролировать использование семафоров и мьютексов.

**8. Мониторы. Решение задачи Производитель-Потребитель с помощью мониторов.**

Монитор — конструкция языка программирования, поддерживающая управляемый доступ к разделяемым данным. Монитор инкапсулирует:

1. Разделяемые критические данные.
2. Функции, использующие разделяемые данные.
3. Синхронизацию выполнения параллельных потоков, вызывающих указанную функцию.

* Доступ к данным, располагающимся в мониторе, реализуется только из его функций.
* Код синхронизации добавляется автоматически компилятором.



Для каждого экземпляра монитора в каждый момент времени может выполняться не более одной функции.

Conditional Variables — условные переменные, олицетворяющие наступление некоторого события.

Над CV определены следующие операции:

wait(cv) — дождаться наступления события. На время ожидания блокировка с монитора снимается.

signal(cv) — оповестить один ожидающий поток о наступлении события.

broadcast(cv) — оповестить все ожидающие потоки о наступлении события.

*Решение задачи «Производители-Потребители» с помощью мониторов.*

Monitor CB;

Element Buffer[n]; int start=0, end=0;

Conditional Variable NotFull, NotEmpty;

Produce(el){

        if (Буфер полон) wait NotFull;

       Добавить(el);

       signal NotEmpty;

}

Consume(\*el){

        if (Буфер пуст) wait NotEmpty;

       \*el=Извлечь();

       signal(NotFull);

}

**Мониторы Хаара**if (усл) wait (cv);

При вызове signal:

1. Немедленно запускается ожидающий поток.
2. Поток, пославший сигнал, блокируется на все время, пока выполняется поток, которого он вывел из состояния ожидания.

+ Предсказуемость.

**Мониторы Меса**while (усл) wait (cv);

При вызове signal:

1. Ожидающий поток переводится в состояние «Готов к выполнению». Поток, пославший сигнал, продолжает исполнение.
2. При освобождении монитора поток, получивший сигнал, конкурирует с другими потоками за вход в монитор на равных условиях.

+ Более эффективны (используют меньшее кол-во переменных).

+ Непосредственно поддерживают broadcast.

Если вы получили сигнал, значит, *возможно*, состояние монитора изменилось.

**9. Необходимые и достаточные условия тупика.**

Множество процессов находится в состоянии тупика, если каждый процесс ожидает наступления некоторого события, и это событие может быть вызвано только действиями другого процесса из данного множества.

Mutual Exclusion — по крайней мере, один из запрашиваемых ресурсов является неделимым (должен захватываться в эксклюзивное использование).

Hold&Wait (удержание ресурсов при ожидании) — существует процесс, владеющий некоторым ресурсом и ожидающий освобождения другого ресурса.

No Preemption — ресурсы не могут быть отобраны у процесса без его желания.

Circular Wait (циклическое ожидание) — существует такое множество поцессов {p1,p2,...,pn}, в котором p1 ждет p2, p2ждет p3, ... ,pnждет p1.

**10. Граф PR.**

Включает множество вершин V=P U R:

P={p1, p2, …, pn} — множество процессов.

R={r1, r2, …, rn} — множество ресурсов.

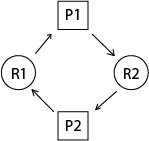
Дуги запросов Pi→Rj — i-й процесс запрашивает j-й ресурс.

Дуги распределения Rj→Pi — j-й ресурс принадлежит i-му процессу.

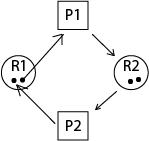
Если PR-граф не имеет циклов, тупиков нет.

Если имеет — *возможно*, тупик существует.

Цикл + тупик:



Цикл без тупика:



**11. Редукция PR-графа.**

PR-граф может быть сокращен за счет процесса, все запросы которого могут быть удовлетворены. При сокращении все ресурсы выбранного процесса освобождаются. Все дуги графа, связанные с этим процессом, удаляются.

В частности,

1. Если граф полностью редуцируем, взаимоблокировка отсутствует.
2. Порядок выполнения редукций не имеет значения.

**12. Предотвращение тупиков.**

1. Mutual Exclusion (делается в первую очередь там, где можно).

Можно построить над ресурсом абстракцию, не требующую эксклюзивного доступа.

− Не всегда возможно.

− Требует доп. затрат ресурсов.

2. Hold&Wait. Идеи:

1. Заставить все процессы запрашивать все необходимые им ресурсы до начала выполнения. − Неэффективное использование ресурсов.
2. При возникновении потребности в ресурсе освобождаются все уже имеющиеся и запрашиваем их обратно вместе с доп. ресурсом. − Неэффективно. − Не всегда реализуемо.
3. Предоставить прикладному программисту неблокирующие функции запроса ресурсов — единственное работающее решение.

3. No Preemption (практически не реализуемо) — можно разрешить ОС передавать ресурсы от одного процесса другому:

1. Заморозить процесс, владеющий ресурсом.
2. Передать ресурс другому процессу.
3. После освобождения ресурса вернуть его исходному владельцу и продолжить выполнение.

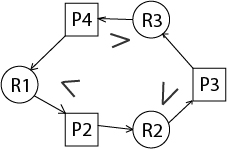
− Чтобы передать ресурс, он должен быть в стабильном состоянии.

− Ресурс может быть зависимым.

Применимо только в очень частных случаях внутри конкретного проекта.

4. Circular Wait. Идеи:

1. Позволить процессам в каждый момент времени владеть только одним ресурсом.
2. Ввести нумерацию ресурсов. Обязать процессы запрашивать ресурсы строго в порядке возрастания их номеров. − Неэффективно.



**13. Избегание тупиков. Алгоритм Банкира.**

Распределить ресурсы таким образом, чтобы не попасть в тупик.

*Алгоритм Банкира.*

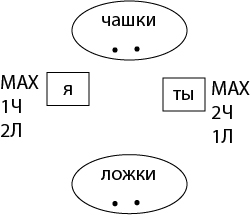
Условия: считаем, что максимальные требования всех процессов к ресурсам известны до начала их выполнения.

Определение: состояние называется безопасным, если для него имеется последовательность процессов {p1,p2,...,pn}, такая, что для каждого Pi ресурсы, которые потребовал Pi, могут быть предоставлены за счет имеющихся не занятых ресурсов и ресурсов, выделенных всем процессам Pj, где j<i.

Перед выделением ресурса проверяем, является ли состояние, в которое мы перейдем после выделения, безопасным. Если новое состояние безопасно, выделяем ресурс. Если нет — ресурс не выделяется, процесс, выполнивший запрос, блокируется.

Алгоритм Банкира можно проиллюстрировать на графе PR. При выполнении запроса:

1. Представляем, что мы его удовлетворили.
2. Представляем, что все процессы запросили максимум ресурсов.
3. Проверяем, может ли PR-граф быть полностью редуцирован. Если да, выделяем ресурс, если нет — блокируем процесс, выполнивший запрос.



1. Я→Ч +
2. Ты→Ч +
3. Ты→Л −
4. Я→Л +

**14. Устранение тупиков. Восстановление после тупика.**

Алгоритм обнаружения — редукция графа PR.

1. При каждом запросе — выделении ресурса.
2. При каждом запросе, который не может быть немедленно удовлетворен.
3. Периодически (например, каждые 2 минуты).
4. Использование косвенных признаков (например, загрузка ЦП за секунду < 20 %).

На практике используется комбинация подходов 2, 3, 4.

1. Уничтожение одного/нескольких/всех процессов, участвовавших в тупике.
2. Перераспределение ресурсов между процессами вплоть до разрушения тупика. Системно почти не применимо. Можно попробовать использовать внутри приложения.
3. Откат одного/нескольких процессов к некоторой контрольной точке.