Взаимодействие параллельных процессов (продолжение).

Блокировки - зло, т.к. к одной крит. секции м.б. очередь блокированных процессов (снижается производительность)
Решение, связанное с очередями:
алгоритм Лампорта - „Булочная";
Это новое решение задачи детектуры об управлении процессами;

алгоритм решает задачу взаимоисключения n процессов

Каждому новому клиенту выдаётся номер, на 1 больший всех ранее выданных номеров. Когда продавец освобождается, он обслуживает клиента с меньшим номером.
Может возникнуть ситуация, когда два клиента приходят одновременно. Что тогда делать?
Или выдаются одинаковые номера, но при обслуживании учитывается клиент, у которого, например, больше номер паспорта.

При решении должен существовать глобальный разделяемый массив текущих номеров. При входе процесс поэлементно проверяет все другие процессы и ждёт каждого, кто имеет меньший номер.
У процессов нет паспорта, поэтому они идентифицируются их pid;

Решение сформулировано Лампортом для распределённой системы с n процессорами;

```
shared boolean choosing[n]; // 2 разделяемых массива:
shared int num[n];          // массив номеров, т.е. средство взаимодействия для i-го процесса
...
for(j=0;j<n;j++)
{
    num[j]=0;
}
...
/* choose a number */
choosing[i]=TRUE;
num[i]=max(num[0], ..., num[n-1])+1;
choosing[i]=FALSE;
/* for all other processes*/
for (j=0;j<n;j++)
{
    /* Wait if the process is currently choosing */
    while(choosing[j]){/* nothing */}
    /* Wait if the process has a number and comes ahead
       of us */
    while (num[j]<>0 and ((num[j],j)<(num[i],i))) do
        /* nothing */
}
    /*critical section*/
    num[i]=0;
```

→ лексикографическое отн-е; если у процессов одинаковые номера, в крит. участок войдёт

процесс с меньшим идентификатором.

Лексикографический порядок:
$(a,b) < (c,d)$ – истина, если $a < c$ или $a = c$, но $b < d$

Если процесс не пытается войти в свой крит. участок, то его эл-т массива num[i] равен 0;

Здесь нет блокировки, но есть опоздание;

Есть ещё Black-White Bakery Algorithm;

Скорее всего возьмём классический алгоритм „Булочная", когда будем писать на сокетах;

В Linux есть много реализаций test_and_set

Помимо test_and_set часто используемой командой является compare_and_swap – сравнение и перестановка

⇦ опасная посл-ть действий

compare_and_swap – неделимая команда, т.е. в рамках одной неделимой команды выполняются два совершенно разных действия – сравнение и перестановка;

```
compare_and_swap(int *reg, int oldval, int newval)
{
    int old_reg_val = *reg;
    if (old_reg_val == oldval)
        *reg = newval;
    return old_reg_val;
}
```

В UNIX и Windows есть систем. вызов Mutex (mutual exclusion)

Семафоры против мьютексов.

1) Mutex имеет владельца – процесс, который захватил Mutex; только этот процесс может его освободить (unlock);

   А у семафоров нет владельца – любой может разблокировать семафор:

P1:	P2:
...	...
P(S)	V(S)
...	...

   Это осознанно заложено в семафоры;

2) В отличие от семафоров, на мьютексах определена инверсия приоритетов (priority inversion);

3) Процесс, удерживающий мьютекс, не м. б. удалён / убит (случайно).

   На семафорах это не определено;

   Зато на мьютексах определены соотв. флаги, заставляющие ОС отслеживать текущую ситуацию;

Процесс, удерживающий семафор, м.б. завершён безусловно, а с мютексом это невозможно;

Семафоры в любой ОС всегда существуют и рассматриваются отдельно от всех остальных средств блокировки

Проблема спящего парикмахера (возникает вск):

У парикмахера есть 1 кресло и в приёмной имеется несколько стульев. Когда парикмахер заканчивает работу, клиент уходит и парикмахер идёт в приёмную, чтобы посмотреть, есть ли там ожидающие клиенты.
Если есть, то он приглашает одного и сажает его в кресло.
Если ожидающих клиентов нет, то он возвращается на рабочее место и спит.
Каждый новый клиент сначала смотрит, что делает парикмахер.
Если он спит, то клиент будит парикмахера и садится в кресло.
Если он работает, то клиент идёт в приёмную.
Если есть свободный стул, он садится и ждёт своей очереди.
Если свободного стула нет, клиент уходит.
Проблема конечности очередей: они не могут быть бесконечными.

Основываясь на простом анализе, казалось бы, такой подход гарантирует, что парик-

мастерская будет работать правильно.

Однако на деле существуют несколько конфликтных ситуаций, которые демонстрируют общие проблемы планирования.

Все проблемы связаны с тем, что действия парикмахера и клиентов асинхронные (выполняются с собственной скоростью и не связаны с действиями других клиентов/парикмахера).

Пример 1: клиент может войти, увидеть, что парикмахер работает и отправиться в приёмную. Но, пока он идёт, парикмахер может закончить свою работу и сходить в приёмную, чтобы проверить наличие клиентов. При этом парикмахер может сделать это быстрее клиента. Он придёт в приёмную, увидит, что клиентов нет, вернётся на своё рабочее место и заснёт. А клиент в это же время дойдёт до приёмной, сядет на стул и будет ждать. Итого: парикмахер спит, потому что в приёмной нет клиентов, а клиент сидит в приёмной и ждёт, когда парикмахер закончит работу.

Пример 2: два клиента могут прийти в парикмахерскую в одно и то же время, когда в приёмной есть один единственный свободный стул. Сначала они проверяют, работает ли парикмахер (пусть он работает),

а потом идут в приёмную и пытаются занять единственный стул.

Часто эту задачу приписывают Дейкстре (1965 г.)

↳ Стоит отметить, что отметить, что это очень важные годы, когда появилось 3-е поколение ЭВМ и мощности вычислительных систем поставили перед разработчиками ряд важнейших проблем, которые до этого перед ними не возникали.

Решение задачи не предлагается, т.к. самое интересное - сама проблема;

Взаимодействие процессов в распределённых системах.

Это системы с раздельной памятью, т.е. каждый хост (узел) имеет собственную память.

(А у нас в комах - SMP архитектура, где все процессоры работают с общей памятью)

Семафоры и программные каналы (pipe) не могут использоваться, т.к. они предполагают общую память.

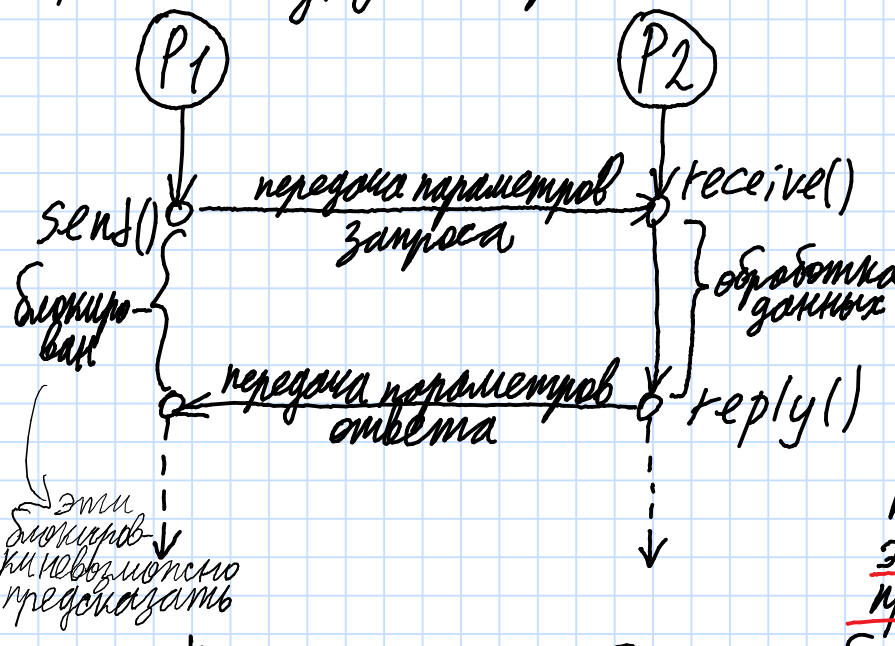
Взаимодействие в таких распр-х системах возможно только путём передачи сообщений.

Для этого нужны соответствующие сист. вызовы. В общем виде это send_message и receive_message.

При этом в такой распр-й ОС возможны ситуации, связанные с взаимоисключением и синхронизацией.

В англоязычных статьях всё называется синхронизацией, но этот момент требует уточнения

Часто возникает ситуация, когда один процесс заинтересован в рез-те работы другого процесса. Пример:

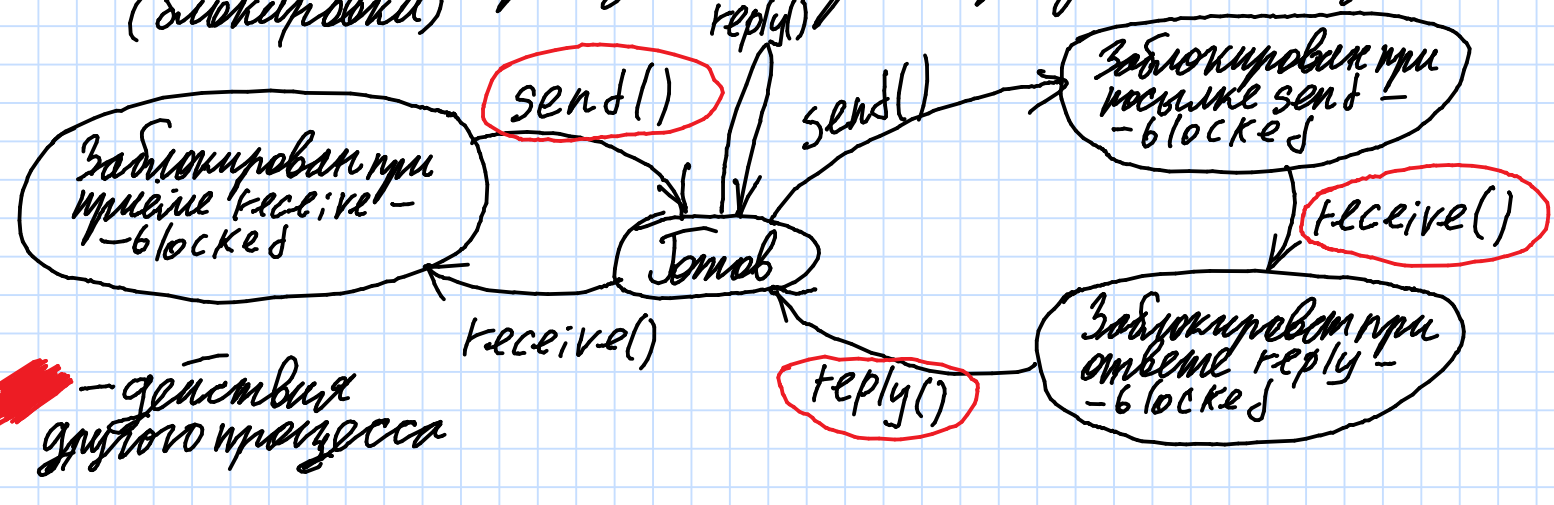


В Linux есть send_msg, receive_msg

Синхронизация: процесс ждёт получения данных, в которых он заинтересован. Эти данные формирует для процесса другой процесс. Без этих данных процесс не может продолжить своё выполнение.

⟹ Возникает проблема блокировки;

3 состояния процесса при передаче сообщений (блокировки):



Процесс блокируется при посылке, когда он вызывает сист. вызов send, а процесс, которому предназначено это сообщение, не готов его принять. Как только адресат вызовет receive, процесс, отправивший сообщение, будет заблокирован при ответе до тех пор, пока им не будет получен ответ.

Процесс, получивший запрос, переходит к его обработке. Когда обработка завершается, он вызывает сист. вызов reply и отправляет ответ процессу, запросившему обслуживание.

Если процесс вызвал receive, а другой процесс ещё не выполнил send, то он будет заблокирован при приёме. (ожидание сообщения, которое ещё не отправлено)
Все эти блокировки возможны в случае протокола взаимодействия, обеспечивающего надёжную передачу сообщений с подтверждением приёма.

Такая ситуация с большим кол-вом блокировок-не редкость (очень характерна для микроядерной архитектуры)

Пусть есть отдельно стоящие машины и программа UNIX также.
Большие программы обычно разбиваются на несколько файлов. При этом изменения, вносимые в один, предполагают его перекомпиляцию.
При этом есть возможность редактировать файлы на одной машине, а компилировать на другой.

Программа также после её запуска проверяет время последнего изменения всех исходных файлов и соответствующие модификации объектных файлов.

Если время последнего изменения исходного файла меньше времени изменения объектного файла, так ничего не делает. Если время изменения объектного файла меньше времени редактирования исходного файла, так считает, что файл необходимо перекомпилировать.

На каждом компьютере есть свой кварцевый генератор (локальные часы). Даже если всё идеально, этот генератор имеет ограниченную точность ($\approx 10^{-5}$)

Пусть таймер генерирует 60 прер-й (импульсов) / сек, т.е. 216 000 прер-й / час. Учитывая указанную точность, это означает, что для каждой конкретной машины может находиться в диапазоне 215 998 ÷ 216 002 тиков в час.

<имя>.c — 1224

<имя>.obj — 1223

требуется перекомпилировать

продолжим в след. раз