Лекция 6. Организация параллельных процессов.

1. Организация параллельных процессов.

В общем случае организация параллельных вычислений подразумевает организацию параллельных процессов на некой сети вычислителей. Собственно организация параллельных процессов включает:
1) Создание нескольких последовательных процессов.
2) Организацию обмена данными между процессами.
3) Синхронизацию работы созданных последовательных процессов.
4) Использование процессами разделяемых ресурсов.

2. Создание нескольких последовательных процессов.

Существуют три способа создания нескольких последовательных процессов:
а) Запуск пользователем нескольких программных кодов на некотором количестве вычислителей. На нескольких ВС запускаются пользователем вручную заранее подготовленные коды – не всегда одинаковые и совместимые, но всегда машиннозависимые.
Примеры: 1) comp1>main1.px; … comp10>main10.px; 2) server>mpirun –np 8 main.px (сокеты TCP/IP)
б) Запуск из головной программы нескольких новых процессов. В головной программе используются стандартные функции ОС UNIX для создания и работы с дочерними процессами. Среди основных функций используются ***fork*** (создание полной копии процесса), ***execl*** и ***execv*** (запуск исполняемого кода из файла), ***system*** (запуск команды shell), ***wait*** (ожидание завершения работы потомков). Процесс в UNIX – самостоятельная исполняющаяся в данный момент программа. Все процессы в UNIX уникальны и имеют свой PID. Исходный родитель – процесс с PID=0 – запускает все остальные служебные процессы с помощью функций fork, exec, system. Такая же возможность предоставляется и пользователю.
в) Создание ветвей последовательного процесса. Создание ветвей одного и того же процесса с помощью библиотеки threads (задач, нитей), содержащей функции pthread_create (создание новой ветви), pthread_join (ожидание завершения ветви) и др.
г) Гибридный способ, сочетающий несколько предыдущих.

3. Основные способы обмена данными между ними.

1) обмен сигналами (функции kill – посылка сигнала процессу, signal – вызов функции пользователя при получении сигнала с заданным номером);
2) обмен с помощью разделяемых файлов (функция блокировки записи lockf());
3) обмен с помощью каналов (функции pipe, read, write; два файловых дескриптора: для чтения и для записи);
4) обмен с помощью очередей сообщений (функции msgget, msgsnd, msgrcv, msgctl – порции данных одинаковой длины);
5) разделяемые сегменты общей памяти (функции shmget - создание на сервере, shmat - присоединение, shmdt - отсоединение, shmctl - удаление);
6) сокеты TCP/IP и их аналоги.

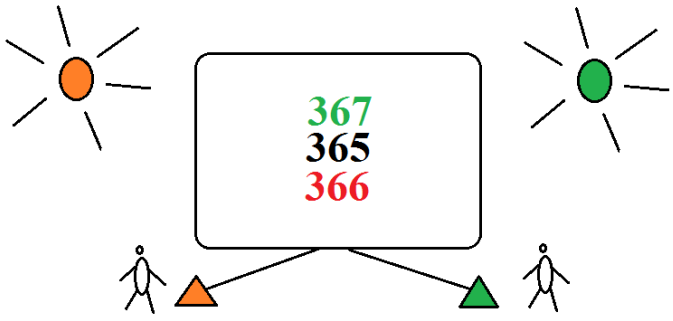
4. Синхронизация работы последовательных процессов.

При организации параллельных процессов (например, решении задачи на нескольких взаимодействующих вычислителях) существуют две фундаментальные проблемы:
1) возникновение тупиков; 2) недетерменированность параллельных вычислений.
Пример – ***задача Дейкстры “Обедающие философы”:***
В парке гуляют философы. Там же стоит беседка с одним входом. В нее философы заходят время от времени поесть. При входе дворецкий регулирует проход. В беседке - стол, накрытый на пять персон. На столе - блюдо с макаронами, 5 тарелок и 5 вилок.
Общая схема питания (Схема 1): сесть за стол, двумя вилками положить макароны, затем одной есть и выйти. Второй вилкой можно пользоваться, если кто-то не ест. ***Трудности:*** Пока философов меньше 5-ти, все ОК (все поедят). Если же одновременно придут все 5 – ***тупик*** (никто не может начать есть).
Схема 2. Учтем конечность времени пользования вилками.
Вар. 1. Дворецкий не пускает к столу одновременно более 4-х человек. Тупика не будет, но есть неудобство – очередь перед столовой. Философ, стоящий в конце очереди рискует не поесть никогда. Например, если очередь типа стэка (первый пришел – последний ушел) => ***первый голодный***. Если первый пришел – первый ушел => есть ***проблема кормления привелегированных особ***. В обоих случаях нет гарантии обслуживания абсолютно всех философов.

Вар. 2. Добавим конечность времени обслуживания одного философа с помощью правила: если все вилки заняты и никто не ест, то все вилки кладем на стол и ждем некоторое время. Затем повторяем попытку. В этом случае будет либо ***драка***, либо опять ***тупик***.
Вар. 3. Если у всех по одной вилке, а в тарелке пусто, то тот, кто сидит на 1-ом месте кладет свою вилку на некоторое время. Тогда 1-ый вилку положил, 2-ой взял, положил макароны, вилку вернул, поел и ушел. На место 2-го пришел новый философ. Если опять все взяли вилки одновременно, то ситуация повторяется. => ***Первый не поест никогда***.
Вар. 4. Передвигать маркер места соседу. Тогда ситуация улучшится. Есть и другие варианты.
Иллюстрацией недетерменированности параллельных процессов здесь служит то, что мы не знаем, в каком порядке обслуживают философов.
Вывод: важнейший вопрос организации параллельных процессов – синхронизация их взаимодействия, особенно при пользовании общими ресурсами. Как это конкретно делается? См. ниже.
Существуют ***2 способа синхронизации*** последовательных процессов:
а) передача сообщений;
б) использование общих данных.
Следует также учесть: при использовании нескольких процессов нет единого “глобального” времени, т.е. нельзя сказать, что событие А на процессоре К произошло раньше события Б на процессоре М.
Поэтому алгоритм пишется в предположении, что ***различные процессы работают с различной не известной априори и меняющейся со временем скоростью.***

5. Общие разделяемые ресурсы и методы их совместного использования.
Разделяемые ресурсы – те, что доступны к использованию нескольким процессам. Например, это общая память, каналы связи, файлы и др.
Примеры часто встречающихся ошибок, связанных с разделяемыми ресурсами:
1) одновременная запись и чтение в общую память несколькими процессами;
2) использование асинхронно получаемых данных до их реального получения;
3) модификация асинхронно передаваемых данных до их реальной отправки;
4) одновременная запись в файл;
5) вызов из нескольких процессов одной и той же “нереентерабельной” функции, т.е. модифицирующей свои внутренние статические данные (например, printf).
Эти проблемы решаются при использовании ***низкоуровневых средств синхронизации – синхронных каналов связи и семафоров***.

6. Решение проблемы синхронизации и разделения ресурсов.
Проблема разделения ресурсов (пример). Подсчет событий двумя наблюдателями:
лампа 1 – человек 1 – доска (365) – человек 2 – лампа 2
Одновременно: лев. лампа -> лев. наблюдатель -> 366; прав. лампа -> прав. наблюдатель -> 366. Правильно: 367.



Q: Reg+Read+Sum+Write

Почему так: процесс чтения числа, прибавления 1 и записи исправленного числа разделены по времени. Если эти действия объединить, то ошибки не будет.
Введем следующие понятия:
Изолированный последовательный процесс – последовательность действий, выполняемых независимо от окружения.
Слабо связанные последовательные процессы – изолированные процессы, которые не взаимодействуют между собой за исключением редких и коротких моментов времени.

Параллельный процесс – совокупность слабо связанных последовательных процессов, которые необходимо организовать для выполнения поставленной задачи.
Критический интервал (КИ) – последовательность действий, выполнение которой может быть приравнено к одной непрерываемой операции.
Необходимо защитить критический интервал от одновременного вхождения в него двух процессов.
Соглашения для решения этой задачи:
1) существуют 2 циклических процесса, выполняющих время от времени КИ;
2) внутри КИ каждый процесс находится произвольное конечное время;
3) вне КИ процессы могут работать бесконечно долго;
4) процессы могут иметь любое число общих переменных;
5) процессы могут читать и записывать значения любых, в том числе общих переменных в любое время без ограничений;
6) результат одновременной записи в одну ячейку: либо x, либо y (целостность данных);
7) результат чтения во время записи: либо старый x, либо новый y (целостность данных);
8) если процессу необходимо войти в КИ, то он должен получить возможность входа за конечное время.
В итоге: неделимы остаются только чтение и запись в память.

Вариант 1 решения задачи – безопасный (без взаимной блокировки), детермин. послед. вхождения в КИ: 1,2,1,2,...	
1-ый процесс	***2-ой процесс***
int next=1;	
while(1){	while(1){
<any actions>	<any actions>
while(next==2);	while(next==1);
{КИ}	{КИ}
next=2;	next=1;
}	}
Вариант 2 (Деккер, 1968 г.) – громоздкий, код неодинаковый, неизвестно, как обобщить на p процессов	
1-ый процесс	***2-ой процесс***
int next=1, C1=1, C2=1;	
while(1){	while(1){
<any actions>	<any actions>
A1: C1=0;	A2: C2=0;
L1: if (C2==0){	L2: if (C1==0){
if (next==1) goto L1;	if (next==2) goto L2;
C1=1;	C2=1;
while (next==2);	while (next==1);
goto A1;	goto A2;
}	}
{КИ}	{КИ}
next=2;	next=1;
C1=1;	C2=1;
}	}

Вариант 3 (Дейкстра, 1965) – использование семафора.
Идея: объединение в одну операцию чтения переменной специального вида, ее модификации и передачи управления в зависимости от ее нового значения. Позволяет иметь единый компактный код и работать с несколькими процессами.
Семафор – неотрицательная целая переменная, доступ к которой возможен только с помощью двух неделимых операций Signal и Wait. Результат Signal(S) – увеличение S на 1 (не равен S=S+1). Если S=5, то результат Signal(S) в двух процессах гарантированно дает 7. Результат Wait(S) – уменьшение S на 1 (если S>0), или останов процесса (если S=0). Останов сбрасывается, как только кто-то выполнит функцию Signal(S), и приостановленный процесс (любой) выполнит в свою очередь уменьшение S на 1.
Решение на обоих процессорах:

```
Semaphore S;
while(1){
<any actions>  Wait(S);  {КИ}  Signal(S);
}
```

Реализация семафоров:

1) тяжелые семафоры – механизм доступа к общим объектам ОС (функции semget, semop, semctl);
2) легкие семафоры – mutex, механизм доступа к общим переменным нашей программы.