Взаимодействие параллельных процессов. (продолжение)

Монопольный доступ обеспечивается методами взаимоисключения.

Семафоры: P(S), V(S) – сист. вызовы;
Только ядро ОС может заблокировать/разблокировать процесс;

Семантически набор семафоров определяется как массив, хотя это массив не в полном смысле слова;

В лабе на семафоры надо будет создавать наборы из 3 семафоров;

Проблема с семафорами: они не структурные. Если нарушено соотв-е между захватом и освобождением семафора (примитивом блокировки и примитивом освобождения), это может привести процессы в тупиковую ситуацию или к неправильной пос-ти действий;

⇒ концепция мониторов (решение проблем путём структурирования использования таких средств)

Мониторы – средство более высокого уровня, чем т.к. примитивы ядра;
↳ низкоуровневые средства в распоряжении у процессов

lock и unlock – захват и освобождение;

lock() — unlock()
wait() — post()
wait() — signal()

Опр. Монитор — механизм, унифицирующий взаимодействие || процессов;
Монитор обозначается ключ. словом
м. б. реализован в ОС или как макросредство в ЯП;
Примеры: concurrent Pascal
Монитор — набор процедур и данных, обращаться к которым можно обращаться только с помощью процедур монитора (он защищает свои данные);
Процесс, находящийся в мониторе — процесс, вызвавший процедуру монитора;

Монитор гарантирует, что в каждый момент времени в ней находится только один процесс;

Остальные процессы, заинтересованные в обращении к монитору, ставятся в очередь;

Как правило, монитор оперирует пер-ными типа условие (condition) с помощью двух оп-ций: wait() и signal(); (сист. вызовы)

wait() блокирует процесс, а signal() разблокирует;

Мониторы:

1) Простой – обеспечивает выделение опр. ресурса произвольному числу процессов

```
resource:monitor // по сути процесс захватывает монитор ⇒ его можно рассм-ть как ресурс
var
    busy:logical;
    x:conditional; // нер-кая типа событие; (conditional) — сигнал для ОС о том, что на этой спец. пер-ной будут вып. опр. действия — какие выполняются опр. действ.
procedure acquire;
begin
    if busy then wait(x);
    busy:=true
end;

procedure release;
begin
    busy:=false;
    signal(x);
end;

begin
    busy:=false;
end.
// здесь показан сам монитор (его код)
// здесь не показано взаимодействие || процессов;
```

Когда к монитору обращается процесс для захвата ресурса, вызывается процедура acquire. Если busy=true, то по пер-ной x типа условие вызывается сист. вызов wait().

В рез-те значение лок. пер-ной не меняется;
Если busy = false, то процесс получает доступ к ресурсу и продолжается без задержек; значение busy становится true; происходит захват ресурса;
Если процесс хочет освободить ресурс, он вызывает процедуру монитора release, где busy становится false. После этого вызывается функция signal(), которая разблокирует другой процесс, который находится в очереди к пер-ной типа событие (или к монитору);
Для каждой отдельной ситуации, по которой процесс м.б. переведён в сост-е блокировки (ожидания), назначается своя пер-ная типа условие (по сути это обозначение соответствующей очереди);
wait и signal похожи на P и V — это сист. вызовы;
Семафоры всегда существуют в любой ОС;
Пример:
В UNIX и Windows есть и семафоры, и мьютексе (mutex);
Отличие семафоров от мьютексов: ... а где... почему-то след. тема началась

2) Монитор: кольцевой буфер,
решает задачу производства и потребления;
кольцевой — зацикливается на начало, т.е. он как бы бесконечный;

Для задачи производства и потребления характерны 2 типа процессов:

1) процессы – производители, которые могут только производить единицы данных (и класть их в буфер)
2) процессы – потребители, которые могут только читать данные из буфера;

```
resource : monitor;
var : bufferempty, bufferfull : condition;
      bcircle : array[0..n-1] of <type>
          pos : 0..n ; // текущ. позиция
            j : 0..n-1 ; //заполняется
            k : 0..n-1 ; //освобождается
procedure producer(data:<type>)
begin
      if pos=n then wait(bufferempty);
      bcircle[i]:=data;
      pos:=pos+1;
      j:=(j+1) mod n;
      signal(bufferfull);
end;
procedure consumer(var data:<type>)
begin
      if pos=0 then then wait(bufferfull);
      data:=bcircle[k];
      pos:=pos-1;
      k:=(k+1) mod n;
      signal(bufferempty);
end;
```

```
begin
    pos := 0;
    j := 0;
    k := 0;
end.
```

Алгоритм соответствует решению Дейкстры с 3 семафорами;

Другой процесс сигнализирует об освобождении/занятии ячейки ⟹ разблокируется другой процесс;

Отношение производителей и потребителей — тапу to тапу (в лабе надо 3+ каждого);

3) Монитор: читатели – писатели.

Есть задача производства-потребления, а есть задача читателей – писателей;

Предполагаем наличие 2 типов процессов:

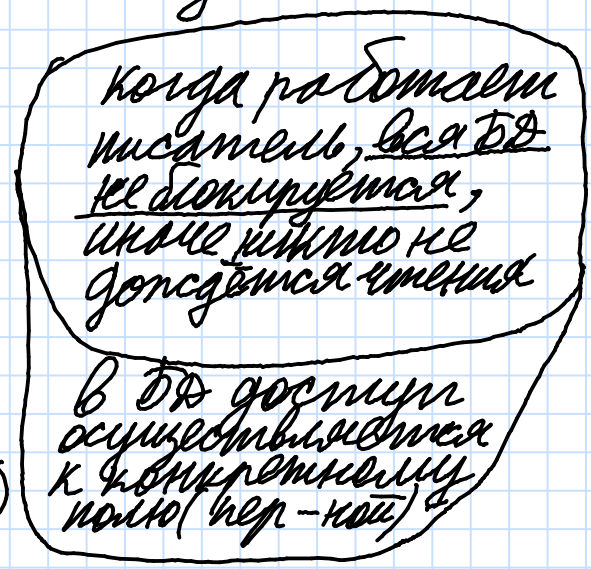
1) <u>процессы</u> – писатели могут изменять данные, поэтому они должны работать в режиме монопольного доступа к разделяемым данным; при этом, если писатель изменяет данные, то к ним не может обратиться ни другой писатель, ни другой читатель;

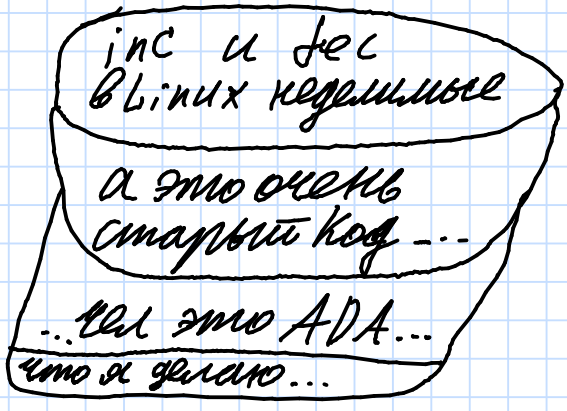
2) <u>процессы – читатели</u> могут только читать данные; при этом они могут

читать параллельно, они друг другу не мешают;

Пример задачи „Читатели-писатели":
Системы массовой продажи билетов (например, на самолёт) на опр. рейс, число и время, где мы резервируем места;
Сначала мы вып. роль читателя, а потом, когда выбираем место и резервируем его, переходим в режим писателя (в другой интерфейс);
Зарезервированное место помечается как условно занятое;

```
resource monitor; // код по Дейтелу
 var
      nr: integer; // читатели
     wrt: logical; // писатель
   c_read, c_write: condition;
procedure startread
                          очередь (ф-ция из ADA)
begin
    if wrt or turn(c_write) ⊛
        then wait(c_read);
    nr := nr+1;
    signal(c_read);
end;
procedure stopread
begin
    nr := nr-1;
    if nr = 0 then signal(c_write);
end;
```





```
procedure start write
 begin
     if nr>0 or wrt then wait(c_write);
     wrt:=true;
end;
procedure stop write;
begin
     wrt:=false;
     if turn(c_read)
     then signal(c_read)
     else signal(c_write);
end;
begin
     nr:=0;
     wrt:=false;
end.
```

// это только код самого монитора;

(*) startread: если есть активный писатель или в очереди есть ждущие писатели, то читатель блокируется на пер-ной типа условие c_read (условие чтения). Если нет активного писателя и нет ждущих писателей в очереди, то кол-во активных читателей увеличивается и посылается сигнал „условие чтения" („можно читать"). Тогда активизируются другие читатели в очереди ждущих читателей.

Особенность данного монитора: создаётся цепная реакция читателей, т.е. каждый новый активный читатель пробуждает следующих читателей в очереди;

Когда читатель завершает чтение, вызывая stopread, кол-во активных читателей уменьшается на 1. Когда это кол-во становится равным нулю (все прочитали), вызывается ф-ция s;g на /(„можно писать"). В данной реализации предполагается, что читатели, заинтересованные в чтении, должны получить такую возможность, т.е. чтение не „обрывается", когда появляется „пишущий" писатель (он ждёт, пока не останется активных читателей, т.е. пока все не дочитают);

startWrite: вызвав её, писатель проверяет, есть ли активные читатели и активный писатель. Если есть, то писатель переходит в сост-е ожидания. Если нет никого, то писатель захватывает критическую пер-ную, обозначая вход в критическую секцию;

stopWrite: сбрасывается пер-ная активного писателя. Если очередь читателей не пуста, посылается сигнал „можно читать", иначе посылается сигнал „можно писать".

Данные условия предотвращают бесконечное откладывание читателей и писателей;

Ещё одно решение для обеспечения монопольного доступа к критическим ресурсам: на основе очереди, которая сама по себе может обеспечить монопольный доступ к крит. ресурсу.

Это решение предложил Лампорт:
„Алгоритм булочная" (bakery algorithm).
Решение базируется на системе „взять номер".
(take a number)