

Aula 10 – Comunicação Interprocessos

Norton Trevisan Roman
Clodoaldo Aparecido de Moraes Lima

11 de outubro de 2014

Soluções de Exclusão Mútua

- Espera Ocupada (Busy Waiting)
- Primitivas Sleep/Wakeup
- Semáforos
- Monitores
- Troca de Mensagem

Monitores

- Idealizado por Sir Charles Antony Richard Hoare (1974) e Per Brinch Hansen (1973), para evitar erros como a inversão dos locks no mutex
 - Inicialmente implementados em Concurrent Pascal (ou PASCAL-FC), de Per Brinch Hansen
- Primitiva (unidade básica de sincronização) de alto nível para sincronizar processos:
 - Conjunto de rotinas, variáveis e estruturas de dados agrupados em um único módulo ou pacote
 - Os processos podem chamar as rotinas em um monitor quando quiserem

Monitores

- Dependem da linguagem de programação

- O compilador é quem implementa a exclusão mútua
- Existente em JAVA, não em C
- Todo recurso compartilhado entre processos deve estar implementado dentro do Monitor

```
monitor example
  integer i;
  condition c;

  procedure producer ();
  .
  .
  .
end;

  procedure consumer ();
  .
  .
  .
end;
end monitor;
```

Monitores – Execução

- É feita uma chamada a uma rotina do monitor
 - Processos não podem acessar as estruturas internas do monitor a partir de procedimentos declarados fora dele
- Instruções iniciais da rotina (incluídas pelo compilador):
 - Teste para detectar se um outro processo está ativo dentro do monitor
 - Se positivo, o processo que chamou a rotina ficará bloqueado até que o outro processo deixe o monitor
 - Caso contrário, o processo novo executa a rotina no monitor

Monitores – Execução

- Somente um processo pode estar ativo em um monitor em um dado instante
 - Outros processos ficam bloqueados
- Implementado pelo compilador, geralmente por meio de um mutex
 - Reduz a possibilidade de erro do programador
- Basta o programador transformar as regiões críticas em procedimentos de monitores
 - Como o synchronized de java

Monitores

- Meio fácil para conseguir exclusão mútua
 - Mas isso não é o bastante
 - Precisamos também de um meio de bloquear processos quando não puderem prosseguir
 - Ex: quando o produtor encontra o buffer cheio, deve bloquear (independentemente de alguém querer ou não usar o mesmo buffer)
 - A solução se dá via variáveis de condição (Condition Variables)
 - Permitem que threads bloqueiem por conta de alguma condição não satisfeita
 - Monitores implementam essas variáveis e as operações definidas sobre elas

Monitores – Variáveis de Condição

- Operações Básicas: WAIT e SIGNAL
 - wait(condition): bloqueia o processo
 - Quando um procedimento (em um monitor) descobre que não pode continuar, executa um wait em alguma variável condicional
 - O processo que chamou wait bloqueia (liberando o monitor)
 - Se houver algum outro processo impedido de entrar no monitor, ele poderá ser escalonado agora
 - Esse outro processo pode acordar o processo adormecido emitindo um signal para a variável condicional que ele espera
 - signal(condition): acorda o processo que executou um wait na variável condition e foi bloqueado

Monitores – Variáveis de Condição

- Como evitar que dois processos permaneçam no monitor ao mesmo tempo?
 - Precisamos de alguma regra que diga o que acontece após um signal:
 - Pode-se exigir que o processo que emitiu o signal saia do monitor imediatamente → signal será o comando final da rotina do monitor
 - Deixar o processo recém-acordado rodar, bloqueando o outro
 - Deixar o processo que emitiu o signal rodar, e então somente permitir que o outro rode após o primeiro ter saído do monitor.

Monitores – Variáveis de Condição

- Usadas em conjunto com mutexes:
 - Uma thread trava um mutex
 - Ao não conseguir algo de que precisa, espera em uma variável condicional (via wait)
 - O wait é chamado atomicamente e destrava atomicamente o mutex que controlava
 - Outra thread pode então continuar, eventualmente sinalizando a thread bloqueada
 - No produtor-consumidor, elimina a necessidade dos semáforos *empty* e *full*

Monitores – Variáveis de Condição

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;
```

```
procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;
```

Concurrent Pascal (ou PASCAL-FC)

Monitores – Variáveis de Condição

- Variáveis de condição não são contadores
 - Não acumulam sinais, como os semáforos
 - Se um sinal é enviado sem ninguém (processo) estar esperando, o sinal é perdido
 - Assim, um comando WAIT deve vir antes de um comando SIGNAL
 - Processos podem verificar se é necessário fazer um signal, inspecionando alguma variável em comum
 - Não é problema, se chamadas a WAIT e SIGNAL estiverem dentro do mesmo monitor

Monitores

- Diferença com sleep/wakeup:
 - Sleep/wakeup falharam porque enquanto um processo tentava dormir, o outro tentava acordá-lo
 - A exclusão mútua automática dentro do monitor garante que se um processo rodar, o outro não estará rodando
- Vida real: Java, via synchronized
 - Embora não tenha variáveis condicionais oferece três procedimentos: wait, notify e notifyall
 - Equivalem a sleep e wakeup, mas dentro de uma zona livre de condição de disputa (bloco synchronized)

Monitores – Variáveis de Condição

● Produtor/Consumidor em Java:

```
public class ProducerConsumer {
    static final int N = 100; // constant giving the buffer size
    static producer p = new producer(); // instantiate a new producer thread
    static consumer c = new consumer(); // instantiate a new consumer thread
    static our_monitor mon = new our_monitor(); // instantiate a new monitor

    public static void main(String args[]) {
        p.start(); // start the producer thread
        c.start(); // start the consumer thread
    }

    static class producer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // producer loop
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // actually produce
    }

    static class consumer extends Thread {
        public void run() { // run method contains the thread code
            int item;
            while (true) { // consumer loop
                item = mon.remove();
                consume_item(item);
            }
        }
        private void consume_item(int item) { ... } // actually consume
    }
}
```

```
static class our_monitor { // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices

    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer[hi] = val; // Insert an item into the buffer
        hi = (hi + 1) % N; // slot to place next item in
        count = count + 1; // one more item in the buffer now
        if (count == 1) notify(); // if consumer was sleeping, wake it up
    }

    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer[lo]; // fetch an item from the buffer
        lo = (lo + 1) % N; // slot to fetch next item from
        count = count - 1; // one fewer items in the buffer
        if (count == N - 1) notify(); // if producer was sleeping, wake it up
        return val;
    }

    private void go_to_sleep() { try(wait()); catch(InterruptedException exc) {} }
}
```

Ao entrar em um método synchronized, o objeto “mon” é bloqueado

Exclusão Mútua em Java

- Semáforos (classe Semaphore):
 - <http://java.dzone.com/articles/dnp-java-concurrency-%E2%80%93-part-4>
 - <http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Semaphore.html>
- Monitores (Synchronized):
 - <http://www.guj.com.br/articles/43>
 - <http://www.artima.com/insidejvm/ed2/threadsynch.html>
- Variáveis de Condição:
 - <http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/Condition.html>

Semáforos e Monitores

- Limitações de semáforos e monitores:
 - Monitores dependem de uma linguagem de programação – poucas linguagens suportam Monitores
 - Ambos são boas soluções somente para CPUs com memória compartilhada (usam a TSL)
 - Não são boas soluções para sistema distribuídos, em que cada processador (ou grupo de processadores) possui sua própria memória, se comunicando via rede
 - Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas
 - Precisamos de algo mais... mensagens!

Soluções de Exclusão Mútua

- Espera Ocupada (Busy Waiting)
- Primitivas Sleep/Wakeup
- Semáforos
- Monitores
- Troca de Mensagem

Mensagens

- Os mecanismos já considerados exigem do S.O. somente a sincronização
 - Asseguram a exclusão mútua, mas não garantem um controle sobre as operações desempenhadas sobre o recurso
 - Deixam para o programador a comunicação de mensagens através da memória compartilhada
- A troca de mensagens é um mecanismo mais elaborado de comunicação e sincronização
 - Exige do S.O. tanto a sincronização quanto a comunicação entre os processos

Mensagens

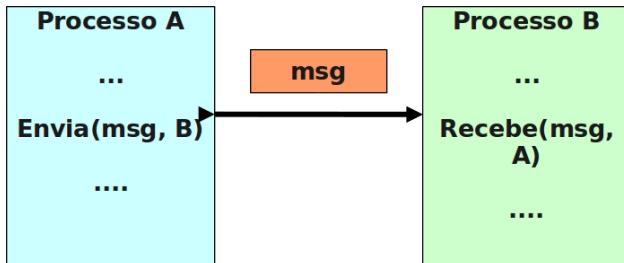
- Esquema de troca de mensagens:
 - Os processos enviam e recebem mensagens, em vez de ler e escrever em variáveis compartilhadas.
 - Sincronização entre processos:
 - Garantida pela restrição de que uma mensagem só poderá ser recebida depois de ter sido enviada
 - A transferência de dados de um processo para outro, após ter sido realizada a sincronização, estabelece a comunicação



Mensagens

- Possui duas primitivas:
 - Envia (Send)
 - `send(destino, &mensagem);`
 - Recebe (Receive)
 - `receive(fonte, &mensagem);`
 - Se não houver mensagem disponível, o receptor pode:
 - Bloquear, até que haja alguma mensagem
 - Retornar com mensagem de erro
 - Implementadas como chamadas ao sistema

Mensagens – Primitivas






O fato de um somente poder receber a mensagem após o outro enviar garante a sincronia entre os processos

Mensagens – Primitivas

- As primitivas podem ser de dois tipos:
 - Bloqueantes: quando o processo que a executar ficar bloqueado até que a operação seja bem sucedida
 - Quando ocorrer a entrega efetiva da mensagem ao processo destino, no caso da emissão
 - Quando ocorrer o recebimento da mensagem pelo processo destino, no caso de recepção).
 - Não bloqueantes: quando o processo que executar a primitiva, continuar sua execução normal, independentemente da entrega ou do recebimento efetivo da mensagem pelo processo destino

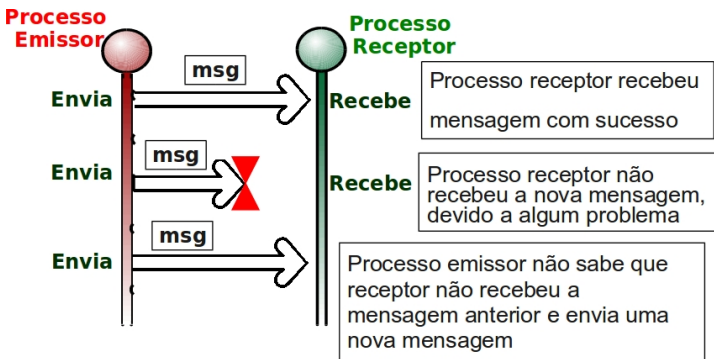
Mensagens – Combinação de Primitivas

- Existem quatro maneiras de se combinar as primitivas de troca de mensagens:

Envia Bloqueante - Recebe Bloqueante		síncrono
Envia Bloqueante - Recebe Não Bloqueante		Semi síncrono
Envia Não Bloqueante - Recebe Bloqueante		
Envia Não Bloqueante - Recebe Não Bloqueante		Assíncrono

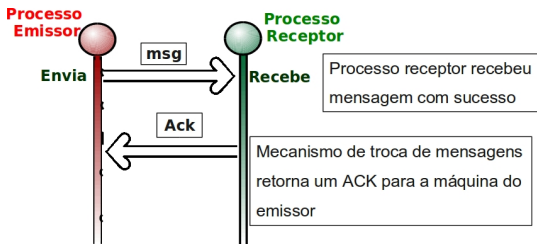
Mensagens – Problemas

- Perda de Mensagens
 - Podem ocorrer por falhas na rede, por exemplo



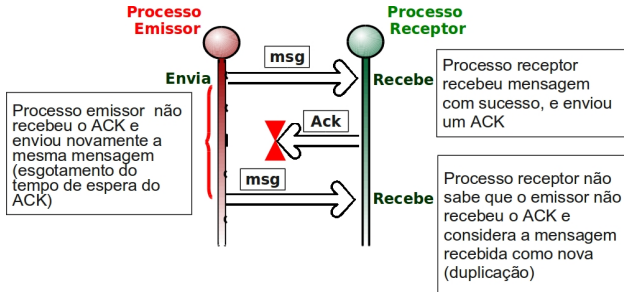
Mensagens – Perda de Mensagem

- Possível solução:
 - O receptor, ao receber uma nova mensagem, envia uma mensagem especial de confirmação (ACK)
 - Se o emissor não receber um ACK dentro de um determinado intervalo de tempo, deve retransmitir a mensagem (laço)



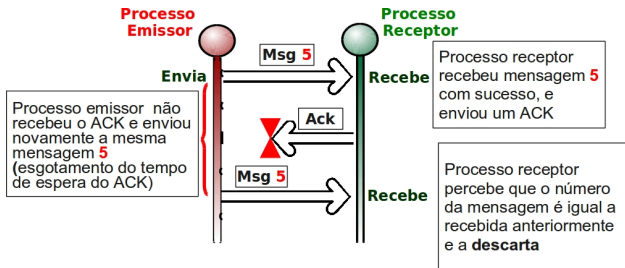
Mensagens – Perda de Confirmação

- O que acontece se a mensagem é recebida corretamente, mas o ACK é perdido?
 - O emissor retransmitirá a mensagem
 - O receptor irá recebê-la duas vezes



Mensagens – Perda de Confirmação

- Possível solução:
 - É essencial que o receptor seja capaz de distinguir uma nova mensagem de uma retransmissão
 - Numerar as mensagens e confirmações:
 - Se o receptor receber mensagem com o mesmo número que alguma anterior, sabe que é duplicata, ignorando-a



Mensagens – Nomes de Processos

- Os processos devem ser nomeados de maneira única, para que o nome do processo especificado no Send ou Receive não seja ambíguo
 - Ex: processo@máquina (normalmente existe uma autoridade central que nomeia as máquinas)
 - As mensagens são então endereçadas aos processos
- Quando o número de máquinas é muito grande:
 - processo@máquina.domínio

Mensagens – Produtor/Consumidor

Assumimos que mensagens enviadas e não lidas são guardadas pelo S.O.

Usamos um número de mensagens igual ao tamanho do buffer

Não há compartilhamento de memória

O consumidor começa enviando N mensagens vazias

```
#define N 100                                     /* número de lugares no buffer */

void producer(void)
{
    int item;
    message m;

    while (TRUE) {
        item = produce_item();
        receive(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);
    }
}




void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* envia N mensagens vazias */
    while (TRUE) {
        receive(producer, &m);
        item = extract_item(&m);
        send(producer, &m);
        consume_item(item);
    }
}
```

/* gera alguma coisa para colocar no buffer */
/* espera que uma mensagem vazia chegue */
/* monta uma mensagem para enviar */
/* envia item para consumidor */

/* pega mensagem contendo item */
/* extrai o item da mensagem */
/* envia a mensagem vazia como resposta */
/* faz alguma coisa com o item */

Mensagens – Comunicação Síncrona



- Existem três mecanismos de comunicação síncrona mais importantes:
 - Rendez-vous 
 - Rendez-vous Estendido 
 - Chamada Remota de Procedimento
 - Ok, ok 

Mas Antes... Cultura geral

- Randevu: sm (fr rendez-vous) bras. Casa de tolerância; bordel, lupanar, p**eiro.
 - Fonte: Michaelis
(<http://michaelis.uol.com.br/>)
 - Hmm, esperamos que não seja nesse sentido



Mas Antes... Cultura geral

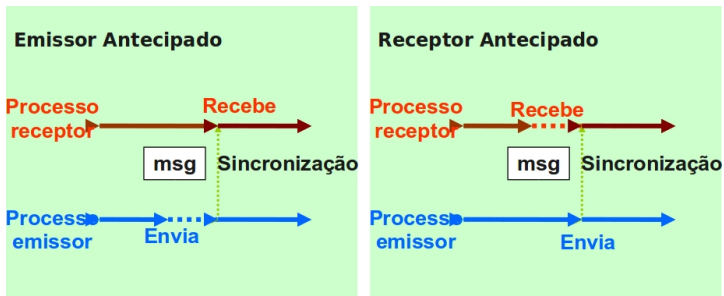
- Randevu: sm (fr rendez-vous) bras. Casa de tolerância; bordel, lupanar, p**eiro.
 - Fonte: Michaelis (<http://michaelis.uol.com.br/>) 
 - Hmm, esperamos que não seja nesse sentido
- Rendezvous: a place appointed for assembling or meeting; a meeting at an appointed place and time
 - Fonte: merriam-webster (<http://www.m-w.com/>)
 - UUUUFFFFF!
 - Esse é o sentido! (Espero!) 

Mensagens – Comunicação Síncrona

- Mecanismo: Rendez-vous
 - Obtido através de primitivas Envia e Recebe bloqueantes colocadas em processos distintos
 - A execução destas primitivas em tempos diferentes faz com que o processo que executar a primitiva antes do outro fique bloqueado até que ocorra a sincronização entre os dois processos, e a consecutiva transferência da mensagem
 - Em seguida, ambos os processos continuarão seu andamento em paralelo
 - Ex.: linguagem CSP

Mensagens – Comunicação Síncrona

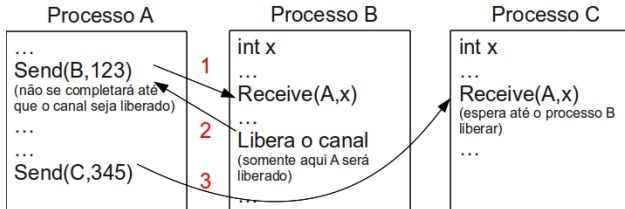
- Mecanismo: Rendez-vous



A transferência da mensagem é copiada diretamente do emissor ao receptor, sem uso de buffer intermediário.

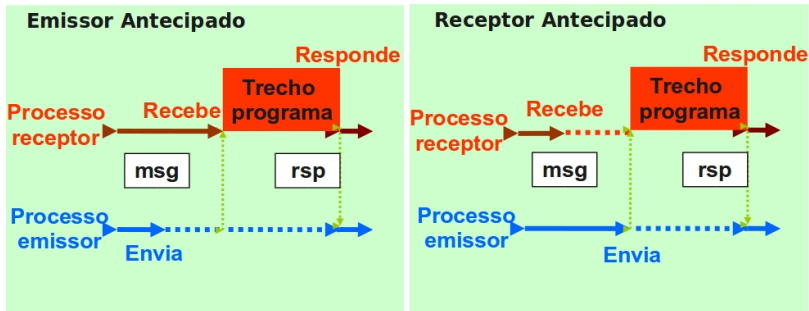
Mensagens – Comunicação Síncrona

- Mecanismo: Rendez-vous Estendido
 - Permite que se execute uma ação após ter recebido um canal de comunicação, mas antes de deixar o outro processo continuar
 - O processo emissor deve esperar que o receptor chegue na parte do código desejada
 - Ex: linguagem ADA



Mensagens – Comunicação Síncrona

- Mecanismo: Rendez-vous Estendido

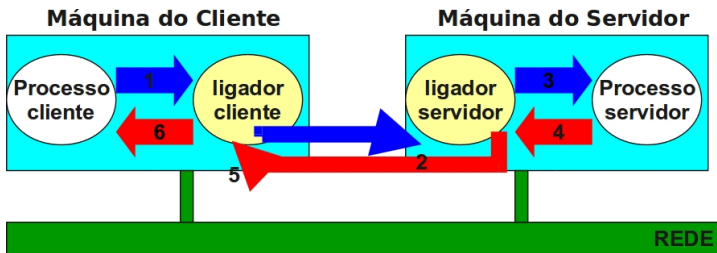


Mensagens – Comunicação Síncrona

- Mecanismo: Chamada Remota de Procedimento (RPC - Remote Procedure Call)
 - Apresenta uma estrutura de comunicação na qual um processo pode comandar a execução de um procedimento situado em outro processador
 - Chamadas remotas a rotinas em diferentes máquinas
 - O processo chamador deverá ficar bloqueado até que o procedimento chamado termine
 - Tanto a chamada quanto o retorno podem envolver troca de mensagem, conduzindo parâmetros
 - Ex: linguagem DP, C (via aplicativo rpcgen)

Mensagens – Comunicação Síncrona

- Mecanismo: Chamada Remota de Procedimento



- (1) e (3) são chamadas de procedimento comuns**
(2) e (5) são mensagens
(4) e (6) são retornos de procedimento comuns

Mensagens – Comunicação Síncrona

- Vantagem da Chamada Remota de Procedimento
 - Cliente e servidor não precisam saber que as mensagens são utilizadas
 - Eles as vêem como chamadas de procedimento locais
- Java: RMI (Remote Method Invocation)
 - Permite que um objeto ativo em uma máquina virtual Java possa interagir com objetos de outras máquinas virtuais Java, independentemente de sua localização
- Outra alternativa: MPI (Message-passing Interface)
 - Sistemas paralelos

Mensagens – Comunicação Síncrona

- Problemas da Chamada Remota de Procedimento
 - Dificuldade da passagem de parâmetros por referência:
 - Se servidor e cliente possuem diferentes representações de informação (necessidade de conversão)
 - Diferenças de arquitetura:
 - As máquinas podem diferir no armazenamento de palavras
 - Ex: long do C tem tamanhos diferentes, dependendo se o S.O. for 32 ou 64 bits

Mensagens – Comunicação Síncrona

- Problemas da Chamada Remota de Procedimento
 - Falhas semânticas:
 - Ex: o servidor pára de funcionar quando executava uma RPC. O que dizer ao cliente?
 - Se disser que houve falha e o servidor terminou a chamada logo antes de falhar, o cliente pode pensar que falhou antes de executar a chamada – Ele pode tentar novamente, o que pode não ser desejável (ex: atualização de BD)
 - Principais abordagens: “no mínimo uma vez”, “exatamente uma vez” e “no máximo uma vez”

Mensagens – Comunicação Síncrona

- Falhas Semânticas – Abordagens:
 - Exatamente uma vez (*maybe*)
 - Toda chamada é executada exatamente uma vez
 - Não há retransmissão de mensagens
 - Se o cliente transmite e o servidor cai, ele não sabe se o servidor processou o pedido antes de cair
 - No mínimo uma vez (*at least once*)
 - O cliente fica retransmitindo o pedido, após timeouts, até que tenha a resposta desejada (ou execute um máximo de retransmissões)
 - O cliente não sabe quantas vezes o procedimento remoto foi chamado (no servidor) – pode ter sido várias

Mensagens – Comunicação Síncrona

- Falhas Semânticas – Abordagens:
 - No máximo uma vez (*at most once*)
 - O cliente fica retransmitindo o pedido
 - Possibilita filtragem de possíveis duplicatas e retransmissão de respostas sem re-executar as operações
 - Se o servidor cair, o cliente saberá do erro, mas não saberá se a operação foi executada
 - Se o servidor não cair, e o cliente receber o resultado da chamada, o procedimento terá sido chamado exatamente uma vez

Outras Referências

- <http://docs.oracle.com/javase/tutorial/networking/overview/networking.html>
- http://en.wikipedia.org/wiki/Two_Generals%27_Problem
- <http://www.firewall.cx/networking-topics/65-tcp-protocol-analysis/134-tcp-seq-ack-numbers.html>