Sistemas Operacionais

Comunicação entre Processos (*Inter-Process Communication - IPC*)

Prof. Arliones Hoeller

arliones.hoeller@ifsc.edu.br

11 de fevereiro de 2014

baseado no material do Prof. Fröhlich em
http://www.lisha.ufsc.br/~guto

Processos Cooperantes

- *Independentes* processo não afeta nem é afetado por outros processos.
- *Cooperantes* processo pode afetar ou ser afetado por outros processos.

- Vantagem de processos cooperantes
  - Compartilhar informações
  - Acelerar computação/processamento
  - Modularidade

- Perigos da cooperação entre processos
  - Corrupção de dados, *deadlocks*, aumento da complexidade
  - Processos precisam sincronizar sua execução

Por que precisamos de IPC?

- Transferência de dados
- Compartilhamento de dados
- Notificação de eventos
- Compartilhamento e sincronização de recursos
- Controle de processos

Mecanismos de IPC

- Mecanismos utilizados para comunicação e sincronização
  - **Troca de Mensagens (*Message Passing*)**
    - Message Passing Interfaces (ex.: MPI)
    - Mailboxes
    - Filas de mensagens (*message queues*)
    - Sockets
    - STREAMS
    - Pipes
  - **Memória compartilhada**: Non-message passing systems

- Exemplos comuns de IPC
  - Sincronização com primitivas como semáforos ou monitores, implementados tanto através de memória compartilhada quanto troca de mensagens.
  - Depuração
  - Notificação de Eventos - UNIX signals

Troca de mensagens

- Um sistema de troca de mensagens não tem variáveis compartilhadas.
- IPC realizada por chamadas a duas primitivas:
  - *send(message)*
  - *receive(message)*

- Se processos *P* e *Q* querem se comunicar, eles precisam::
  - Estabelecer um link de comunicação
  - Trocar mensagens via *send* e *receive*

Questões de Implementação

- Como estabelecer um link de comunicação?
- Um link pode estar associado por mais de um processo?
- Como outros processos ficam sabendo do link?
- Quantos links pode haver entre cada par/grupo de processos?
- Qual a capacidade de um link?
- As mensagens transmitidas por um link têm tamanho fixo ou variável?
- Um link é unidirecional ou bidirecional?

Sistemas de Troca de Mensagens

- Troca de mensagens sobre um link de comunicação

- Há vários métodos para implementar os links e as primitivas *send*/*receive*:
  1. Comunicação Direta ou Indireta (*Naming*)
  2. Comunicação Simétrica ou Assimétrica
  3. *Buffering* Automático ou Explícito
  4. *Send-by-Copy* ou *Send-by-Reference*
  5. Mensagens de tamanho fixo ou variável

Comunicaçao Direta – Internet e Sockets

- Processos precisam se nomear explicitamente:
  - Endereçamento Simétrico
    - send (P, message) – envia ao processo P
    - receive(Q, message) – recebe do processo Q
  - Endereçamento Assimétrico
    - send (P, message) – envia ao processo P
    - receive(&id, message) – recebe de qualquer um; sistema retorna emissor em id
- Propriedades
  - Links estabelecidos automaticamente entre pares
  - Processos precisam conhecer IDs de seus pares
  - Há exatamente um link por par de processos
- **Desvantagem:** um processo precisa saber o nome ou ID do(s) processo(s) que deseja estabelecer comunicação

Comunicação Indireta - Pipes

- Mensagens são trocadas via mailboxes (também chamados de portas)
  - Cada mailbox tem um ID único
  - Processos só se comunicam se compartilham um mailbox
- Propriedades:
  - Link só estabelecido se processos compartilham um mailbox
  - Um link pode estar associado a mais de 2 processos
  - Cada par de processos pode compartilhar vários links
- Posse do link:
  - Posse do processo (mailbox implementado no espaço da aplicação): apenas o proprietário pode receber mensagens por este mailbox. Outros processos só enviam. Quando processo termina, seus mailboxes são destruídos.
  - Posse do sistema: mecanismos providos para criar, deletar, enviar e receber através dos mailboxes. O processo que cria o mailbox o possui, mas pode transferir posse a outros processos.

Comunicação Indireta

- Compartilhamento de Mailbox:
  - P_1, P_2, e P_3 compartilham mailbox A.
  - P_1, envia; P_2 *e* P_3 recebem
    - Quem recebe a mensagem?

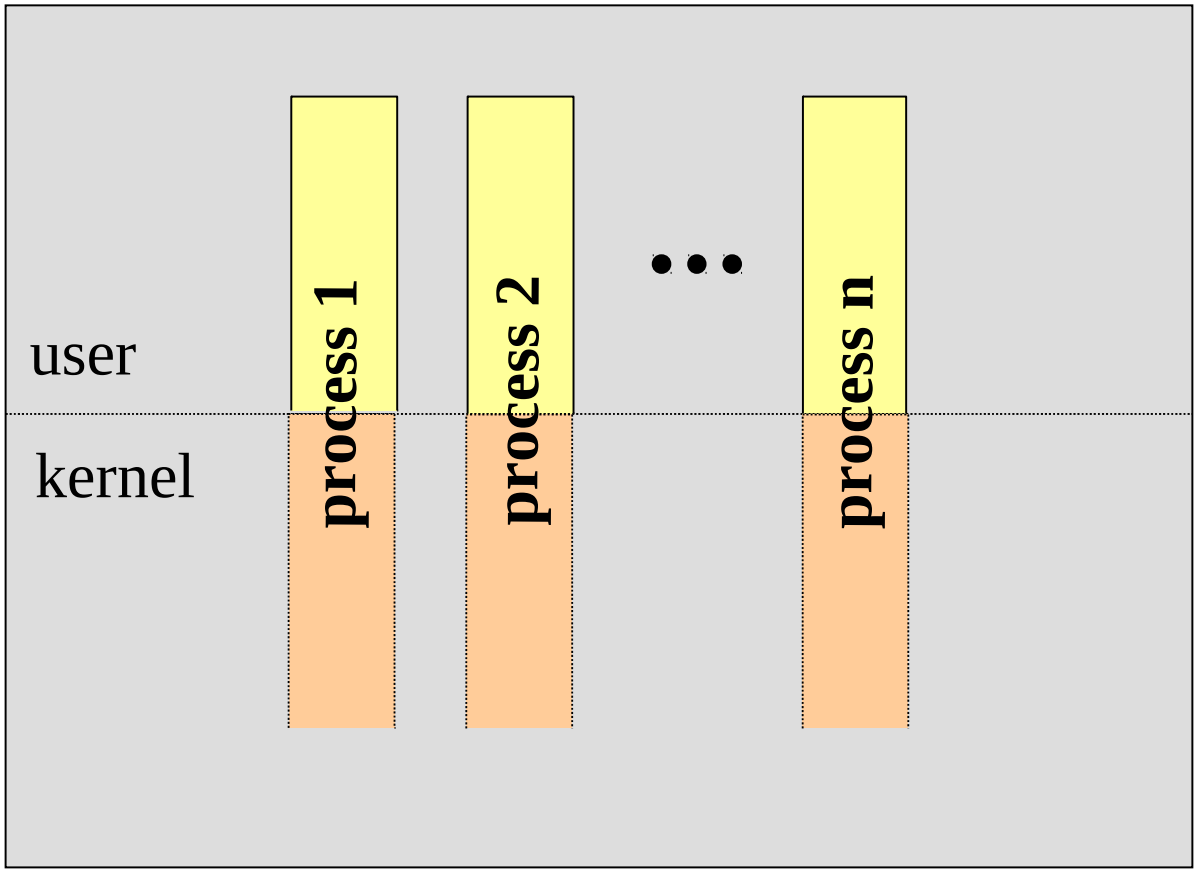
- Soluções
  - Permitir apenas 2 processos associados a um link
  - Permitir que apenas um processo por vez execute *receive*
  - Permitir que o sistema escolha arbitrariamente o recebedor. Emissor é notificado sobre quem recebeu.

Sincronizando o Fluxo de Mensagens

- Troca de mensagens pode ser *bloqueante* ou *não-bloqueante*
  - *send* bloqueante: emissor fica bloqueado até que mensagem seja recebida por mailbox ou processo
  - *send* não-bloqueante: emissor retoma operação imediatamente após enviar
  - *receive* bloqueante: recebedor bloqueia até que uma mensagem esteja disponível
  - *receive* não-bloqueante: recebedor retoma operação imediatamente recebendo ou uma mensagem válida ou a informação de que não havia mensagem disponível

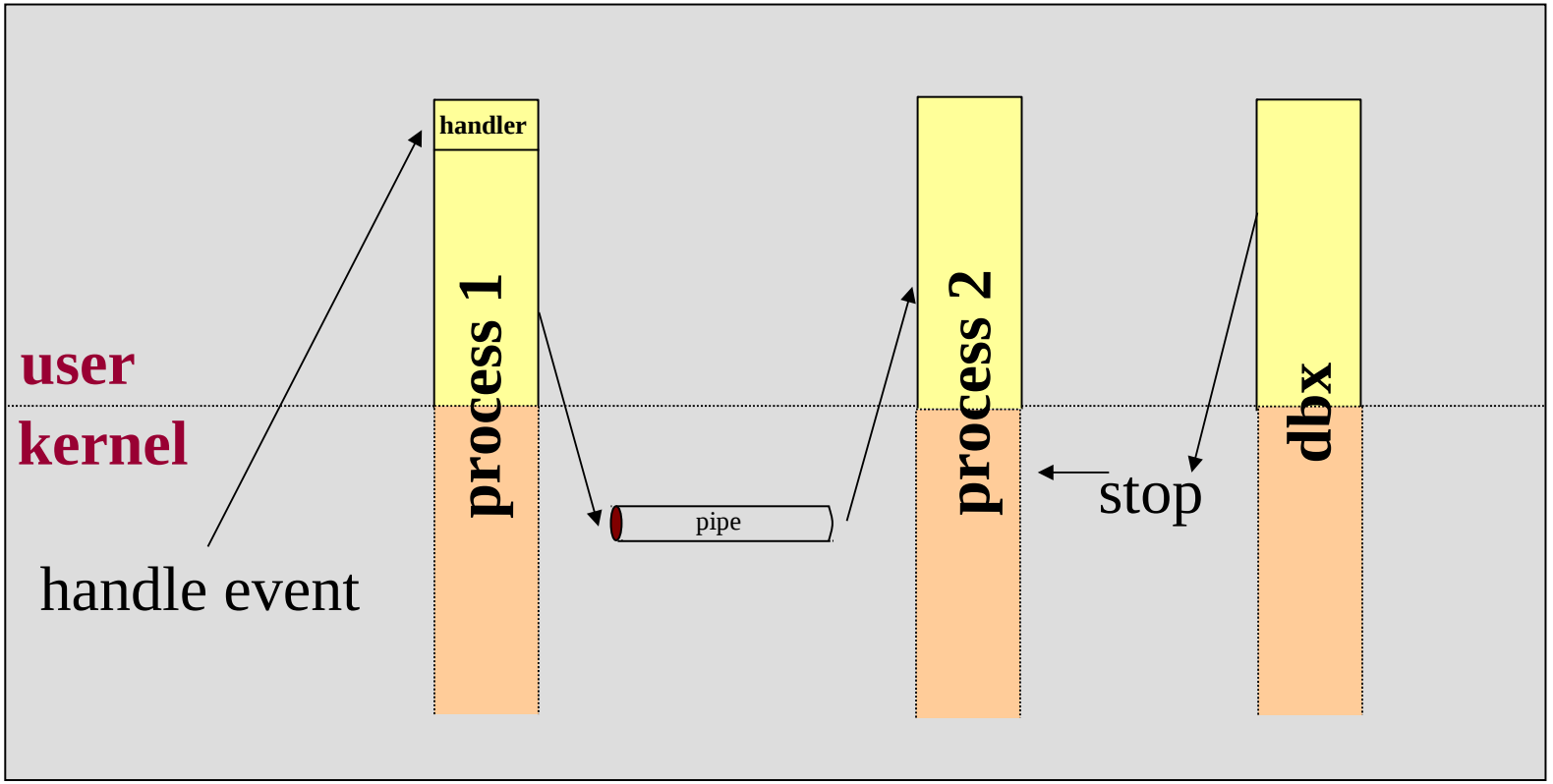
Visão Geral

Domínios de proteçã - (*virtual address space*)



Como os processos podem se comunicar entre si e com o kernel?

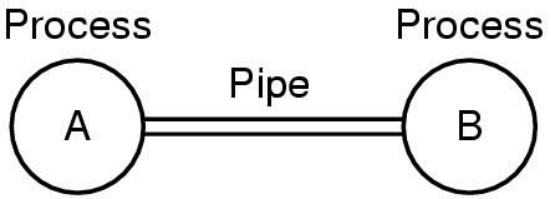
Visão Geral



- IPC no UNIX
  - Signals – notificação de eventos síncrona ou assíncrona
  - Pipes – fluxo de dados unidirecional, FIFO, não-estruturado
  - Process tracing – usado por depuradores (ex.: gdb) para controlar outros processos

UNIX Pipes

- Um **pipe** configura um canal de comunicação unidirecional entre dois processos relacionados



UNIX Pipes

- Um processo escreve no **pipe**, outro processo lê do **pipe**
- Similar a ler/escrever em arquivo
- System calls:

```
int fd[2] ;
pipe(&fd[0]) ;
```

fd[0] possui o descritor para ler do pipe

fd[1] possui o descritor para escrever no pipe

Exemplo Simples

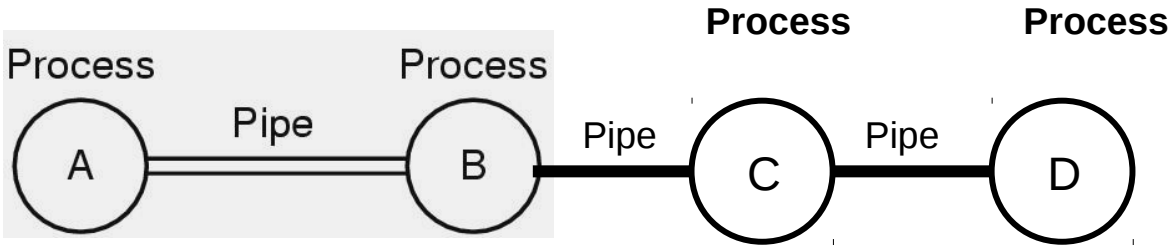
```
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

char *message = "This is a message!!!" ;

main()
{
    char buf[1024] ;
    int fd[2];
    pipe(fd);    /*create pipe*/
    if (fork() != 0) { /* I am the parent */
        write(fd[1], message, strlen (message) + 1) ;
    }
    else { /*Child code */
        read(fd[0], buf, 1024) ;
        printf("Got this from MaMa!!: %s\n", buf) ;
    }
}
```

Criando um *Pipeline*

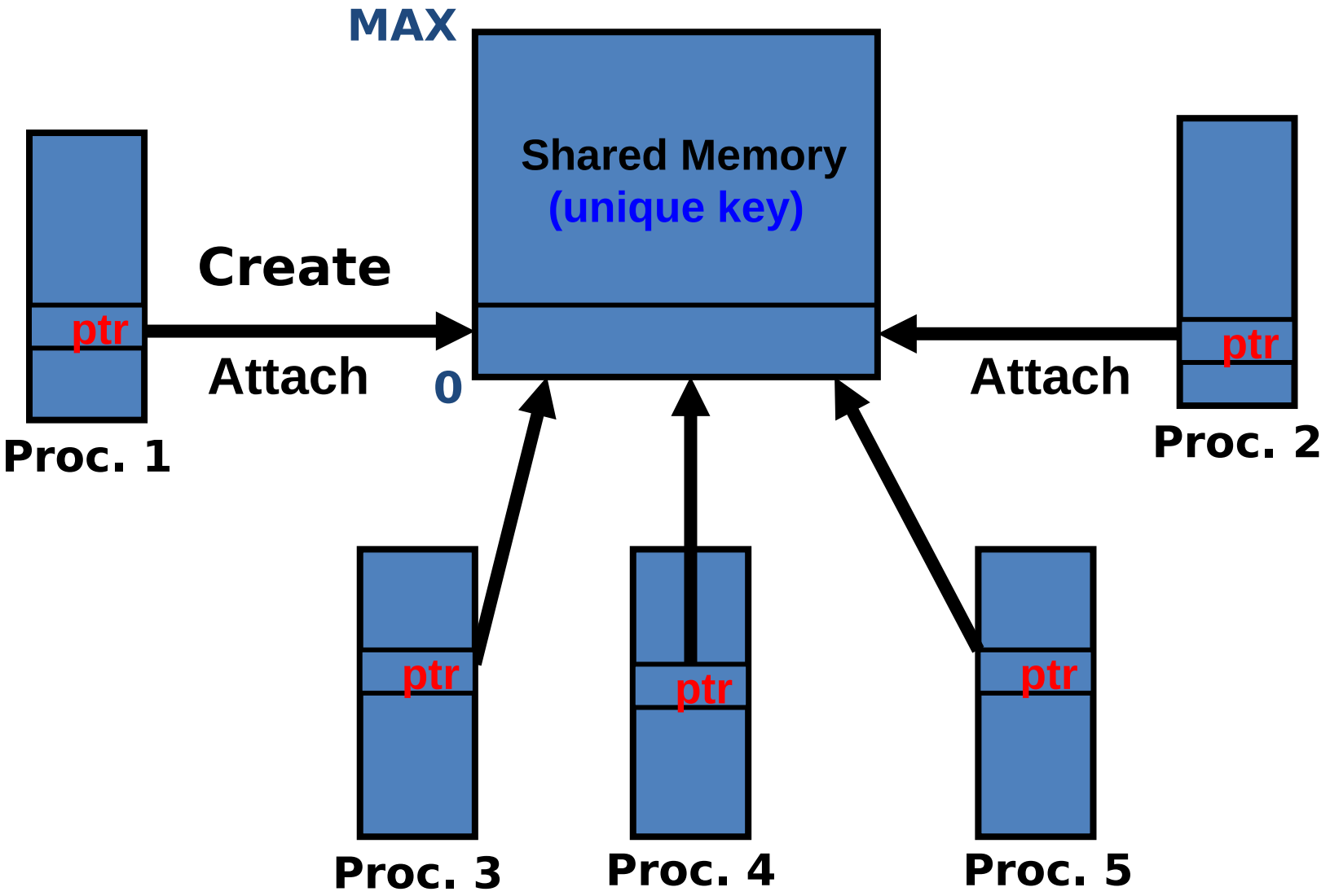
- Às vezes útil para conectar um conjunto de processos



Memória Compartilhada

Sistemas Operacionais

Porção de memória utilizada simultaneamente por processos



Criando Bloco de Memória Compartilhada

```
int shmget(key_t key, size_t size, int shmflg);
```

Exemplo:

```
key_t key;
int shmid;

key = ftok("<somefile>", 'A');

shmid = shmget(key, 1024, 0644 | IPC_CREAT);
```

Anexando e Destacando Memória Compartilhada

```
void *shmat(int shmid, void *shmaddr, int shmflg);
int shmdt(void *shmaddr);
```

Exemplo:

```
key_t key;
int shmid;
char *data;

key = ftok("<somefile>", 'A');
shmid = shmget(key, 1024, 0644);
data = shmat(shmid, (void *)0, 0);

/* use SHM */

shmdt(data);
```

Deletando Memória Compartilhada

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);

shmctl(shmid, IPC_RMID, NULL);
```

Controle de IPC na linha de comando

- ipcs
  - Lista todos objetos IPC do usuário
- ipcrm
  - Remove objeto IPC específico

Sincronização de processos

- Programas concorrentes são executados por múltiplos processos cooperativos que compartilham dados
- O acesso concorrente a dados pode resultar na inconsistência destes dados
- SO precisa prover mecanismos para sincronizar e coordenar processos cooperativos

Produtor X Consumidor

Produtor:

```
shared int counter;
shared char buf[N];

int main()
{
  const int n = N;
  int in = 0;

  while (1) {
    while (counter == n);
    buf[in] = produce();
    in = ++in % n;
    counter++;
  }
}
```

Consumidor:

```
shared int counter;
shared char buf[N];

int main()
{
  const int n = N;
  int out = 0;

  while (1) {
    while (counter == 0);
    consume (buf[out]);
    out = ++out % n;
    counter--;
  }
}
```

Condições de Corrida (*Race Conditions*)

Produtor:

```
  counter++;
  load  R1,[counter]
  inc   R1
  store R1,[counter]
```

Consumidor:

```
counter--
load  R2,[counter]
dec   R2
store R2,[counter]
```

```
0) P: load  R1,[counter]
1) P: inc   R1
2) C: load  R2,[counter]
3) C: dec   R2
4) C: store R2,[counter]
5) P: store R1,[counter]
```

R1	R2	[counter]
5	-	5
6	-	5
6	5	5
6	4	5
6	4	4
6	4	6

Seções críticas

- Seções de programas concorrentes onde dados compartilhados são manipulados
- Condições para execução correta:
  - **Exclusão mútua**: apenas um processo executa uma seção crítica ao mesmo tempo
  - **Progresso**: o próximo processo selecionado para acessar uma seção crítica não pode ser o mesmo que já está nela
  - **Espera limitada**: um processo não pode ser impedido de executar uma seção crítica indefinidamente

Algoritmo de Sincronização I

Processo 0

```
shared int turn;

int main()
{
  while (1) {
    while(turn != 0);

    /* critical */

    turn = 1;

    /* remainder */
  }
}
```

Processo 1

```
shared int turn;

int main()
{
  while (1) {
    while(turn != 1);

    /* critical */

    turn = 0;

    /* remainder */
  }
}
```

- Falha: progresso da execução

Algoritmo de Sincronização II

Processo 0

```
shared int flag[2];

int main()
{
  while (1) {
    flag[0] = 1;
    while(flag[1]);

    /* critical */

    flag[0] = 0;

    /* remainder */
  }
}
```

Processo 1

```
shared int flag[2];

int main()
{
  while (1) {
    flag[1] = 1;
    while(flag[0]);

    /* critical */

    flag[1] = 0;

    /* remainder */
  }
}
```

- Falha: espera indefinida

Algoritmo de Sincronização III (Peterson)

Processo 0

```
shared int turn;
shared int flag[2];

int main()
{
  while (1) {
    flag[0] = 1;
    turn = 1;
    while(flag[1] && turn);
    /* critical */

    flag[0] = 0;
    /* remainder */
  }
}
```

Processo 1

```
shared int turn;
shared int flag[2];

int main()
{
  while (1) {
    flag[1] = 1;
    turn = 0;
    while(flag[0] && !turn);
    /* critical */

    flag[1] = 0;
    /* remainder */
  }
}
```

Sincronização em Hardware

- Instrução *Test and Set Lock* (TSL)

```
int tsl(int * ptr)
{
  int tmp = *ptr;
  *ptr = 1;
  return tmp;
}
```

- Uso

```
shared int lock = 0;
int main()
{
    while (1) {
        while(tsl(lock));
        /* critical */
        lock = 0;
    }
}
```

Semáforos

- Variável inteira acessível através das **operações atômicas** P e V

```
p(s): while(s <= 0);        v(s): s++;
      s--;
```

- Uso

```
shared int mutex;
int main()
{
    while(1) {
        p(mutex);
        /* critical */
        v(mutex);
        /* remainder */
    }
}
```

Implementação de Semáforo

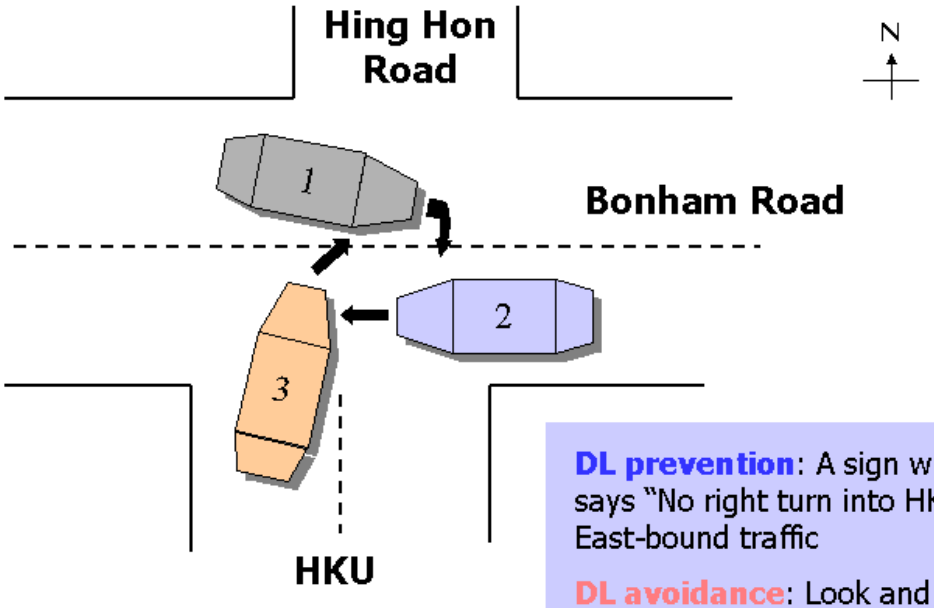
```
class Semaphore
{
public:
  Semaphore(int i) : s(i) {}
  void p();
  void v();
private:
  int s;
  list<Process> l;
};
extern Process * running;
```

```
void Semaphore::v()
{
  if(++s <= 0)
    l.pop()->wakeup();
}
```

```
void Semaphore::p()
{
  if (--s < 0) {
    l.push(running);
    running->sleep();
  }
}
```

Deadlocks

- Um *deadlock* ocorre quando dois ou mais processos estão esperando por um evento que só pode ser gerado pelos processos em espera



DL prevention: A sign which says "No right turn into HKU" for East-bound traffic

DL avoidance: Look and proceed (or not proceed)

DL detection/recovery: Let it happen! ... then recover

Caracterização de *Deadlock*

- Alocação de recursos
  - Requisição => Uso => Liberação
- Condições
  - Exclusão mútua: recursos não podem ser compartilhados
  - *Hold and wait*: um processo retém alguns recursos mas precisa de um recurso que está retido por outro processo
  - Sem preempção: recursos não podem ser “preemptados”
  - Espera circular: deve existir uma espera circular de processos, cada um aguardando por um recursos retido pelo próximo na cadeia

Tratamento de *Deadlock*

- Prevenção
  - Garante que ao menos uma das condições necessárias para caracterizar um *deadlock* nunca acontecerá
- Detecção e recuperação
  - Permite a ocorrência de deadlocks
  - Algoritmo de detecção executa periodicamente
    - Recursos alocados X processos esperando
  - Algoritmo de recuperação executa quando deadlock é detectado
    - Termina processo
    - Preempta recurso (*rollback*)
- Na prática
  - Muito caro, raramente utilizado!