

# Sistemas Hardware-Software

Sinais II: recebimento e concorrência

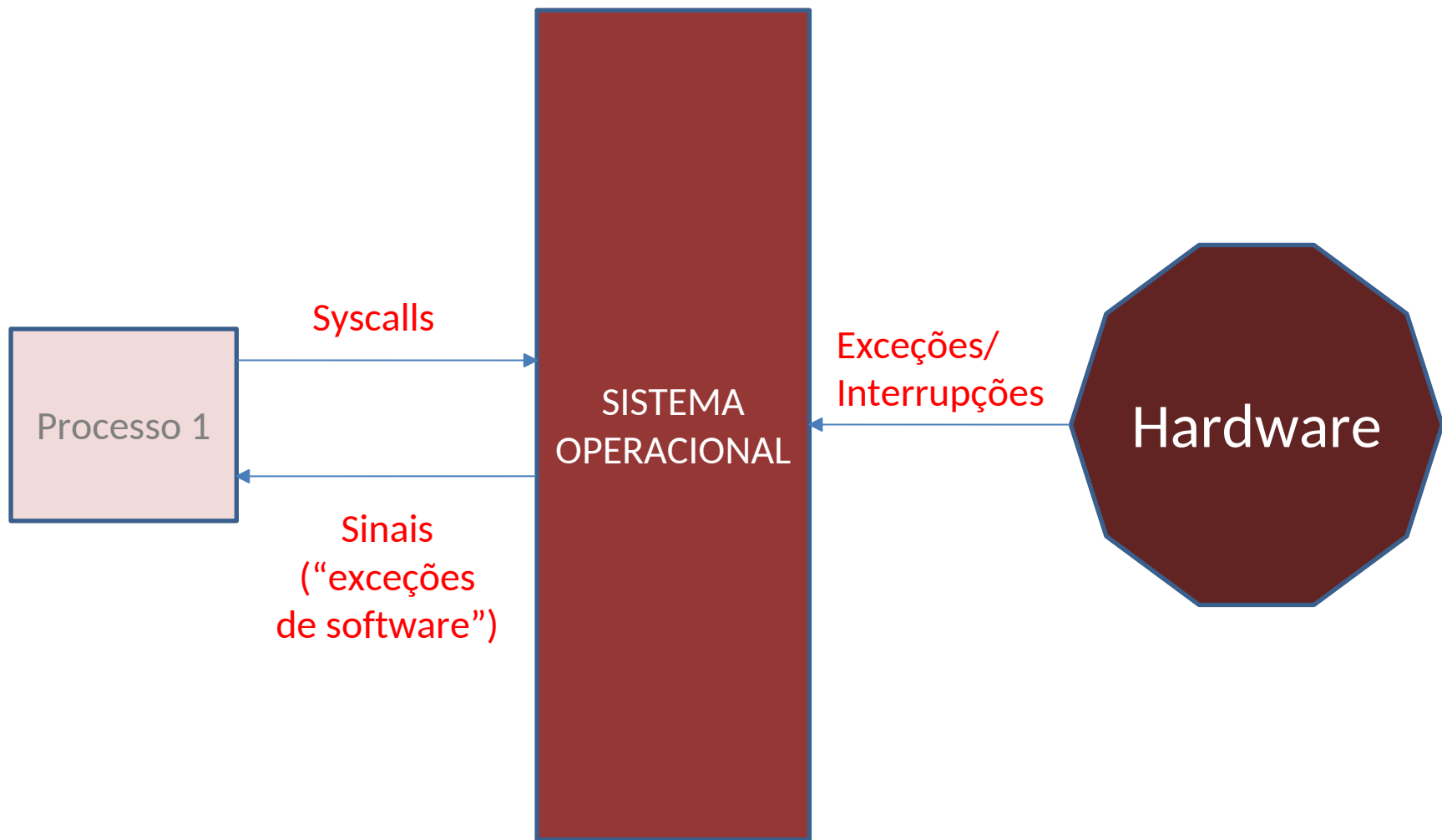
Engenharia

Maciel Vidal  
Igor Montagner  
Fábio Ayres

# Hoje

- Revisão de sinais: wait e kill
- Recebimento de sinais

# Interação do SO com seus processos



# (Alguns) Sinais POSIX

Signal	Default Action	Description
SIGABRT	Terminate (core dump)	Process abort signal
SIGALRM	Terminate	Alarm clock
SIGCHLD	Ignore	Child process terminated, stopped, or continued.
SIGFPE	Terminate (core dump)	Erroneous arithmetic operation.
SIGILL	Terminate (core dump)	Illegal instruction.
SIGINT	Terminate	Terminal interrupt signal. (Ctrl+C)
SIGKILL	Terminate	Kill (cannot be caught or ignored).
SIGTERM	Terminate	Termination signal.

# Exemplos de usos de sinais

- Ctrl+C envia um sinal SIGINT para o processo.
  - o Ele pode ser capturado e fazer com que o programa feche conexões e arquivos abertos, por exemplo.
- O sinal SIGSTOP (SIGTSTP) é usado para deixar um processo em background. Ele fica parado até ser resumido por SIGCONT
- O sinal SIGKILL interrompe um processo imediatamente. Ele não pode ser ignorado.

# Sinais POSIX

Notificação assíncrona enviada para um processo para indicar que algo ocorreu. Principalmente usada para avisar a um processo que

- erros e/ou exceções de hardware ocorreram
- uma condição de sistema mudou
- o usuário quer parar ou finalizar

# Enviando um sinal

- Kernel detectou um evento de sistema, tal como uma divisão-por-zero (SIGFPE) ou término de um processo filho (SIGCHLD)
- Outro processo invocou a chamada de sistema **kill** para explicitamente pedir ao kernel que envie um sinal ao processo destinatário.

# Correção

## **Enviando sinais II (20 minutos)**

1. A chamada de sistema kill



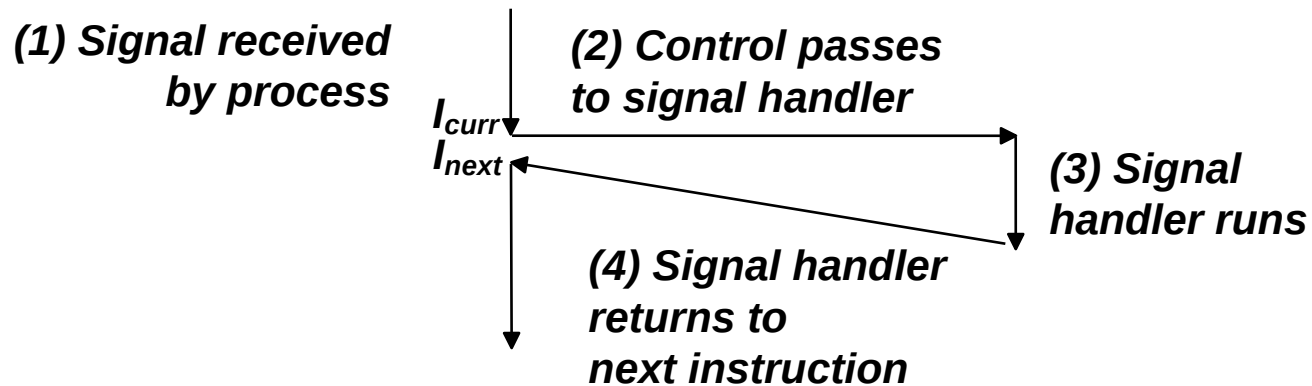
# Recebendo um sinal

O kernel força o processo destinatário a reagir de alguma forma à entrega do sinal. O destinatário pode:

- **Ignorar** o sinal (não faz nada)
- **Terminar** o processo (opcional: core dump)
- **Capturar** o sinal e executar, como usuário, um signal handler

# Captura de sinais

Similar a uma exceção de hardware sendo chamada em resposta a um evento assíncrono



# Recebendo um sinal (Usos)

- Confirmar saída do programa (Capturar)
- Terminar operação que não pode ser interrompida (Ignorar)
- Adicionar tempo limite (Terminar)
- Mudar modo do terminal (Capturar) -> usado por vi, nano, etc

# Recebendo um sinal

```
#include <signal.h>
```

```
int sigaction(int signum, const struct sigaction  
*act, struct sigaction *oldact);
```

```
...
```

Se **act** for non-NULL, a nova ação para o sinal **signum** é executada a partir de **act**. Se **oldact** é non-NULL, a ação anterior é salva em **oldact**.

*oldact is non-null, the previous action is saved in oldact)*

# Recebendo um sinal

```
struct sigaction {  
    void (*sa_handler)(int);  
  
    void (*sa_sigaction)(int,  
siginfo_t*, void *);  
  
    sigset_t sa_mask;  
  
    int sa_flags;  
    void (*sa_restorer)(void);  
};
```

- SIG\_IGN para ignorar
- SIG\_DFL para padrão
- Nome de uma função

Opções de recepção.  
Usaremos 0 sempre aqui.



# Atividade prática

## **Capturando sinais (20 minutos)**

1. Chamada sigaction e seu uso para receber sinais

# Sinais

- Enviados por processos
- Eventos excepcionais externos
- Não carregam informação
- Comportamento padrão:
  - Ignorar, Bloquear, *Handler*
- Uso opcional

# Interrupções (Embarcados)

- Conectados a periféricos
- Entrada de dados
- *Handlers* muito rápidos
- Parte do fluxo do programa
- Essenciais

# Sinais – tentativa 1

```
volatile int flag = 0;

void sig_handler(int num) {
    printf("Chamou Ctrl+C\n");
    flag = 1;
}

int main() {
    int count = 0;
    struct sigaction s;
    s.sa_handler = sig_handler;
    sigemptyset(&s.sa_mask);
    s.sa_flags = 0;
    sigaction(SIGINT, &s, NULL);

    printf("Meu pid: %d\n", getpid());

    while(1) {
        sleep(1);
        if (flag) {
            count++;
            flag = 0;
        }
    }
    return 0;
}
```



# Sinais – tentativa 1

```
volatile int flag = 0;

void sig_handler(int num) {
    printf("Chamou Ctrl+C\n");
    flag = 1;
}

int main() {
    int count = 0;
    struct sigaction s;
    s.sa_handler = sig_handler;
    sigemptyset(&s.sa_mask);
    s.sa_flags = 0;
    sigaction(SIGINT, &s, NULL);

    printf("Meu pid: %d\n", getpid());

    while(1) {
        sleep(1);
        if (flag) {
            count++;
            flag = 0;
        }
    }
    return 0;
}
```

Tenho que incluir essa  
checagem  
em várias partes do programa?

# Sinais – tentativa 1

```
volatile int flag = 0;

void sig_handler(int num) {
    printf("Chamou Ctrl+C\n");
    flag = 1;
}
```

```
int main() {
    int count = 0;
    struct sigaction s;
    s.sa_handler = sig_handler;
    sigemptyset(&s);
    s.sa_flags = 0;
    sigaction(SIGINT, &s, NULL);
```

Erro conceitual: O programa principal espera informações vindas do handler.

```
    printf("Meu programa está rodando\n");
```

Correto: o handler deveria ser autocontido

```
    while(1) {
        sleep(1);
        if (flag) {
            count++;
            flag = 0;
        }
    }
```

```
    return 0;
}
```

# Sinais – tentativa 2

```
volatile int count = 0;

void sig_handler(int num) {
    printf("Chamou Ctrl+C\n");
    count++;
}

int main() {
    int count = 0;
    struct sigaction s;
    s.sa_handler = sig_handler;
    sigemptyset(&s.sa_mask);
    s.sa_flags = 0;
    sigaction(SIGINT, &s, NULL);

    if (count >= 3) return 0;

    printf("Meu pid: %d\n", getpid());

    while(1) {
        sleep(1);
    }
    return 0;
}
```

# Sinais – tentativa 2

```
volatile int count = 0;

void sig_handler(int num) {
    printf("Chamou Ctrl+C\n");
    count++;
}

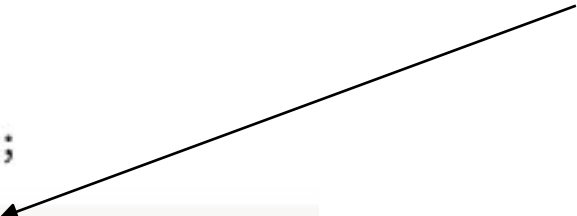
int main() {
    int count = 0;
    struct sigaction s;
    s.sa_handler = sig_handler;
    sigemptyset(&s.sa_mask);
    s.sa_flags = 0;
    sigaction(SIGINT, &s, NULL);

    if (count >= 3) return 0;

    printf("Meu pid: %d\n", getpid());

    while(1) {
        sleep(1);
    }
    return 0;
}
```

E se o código já tiver passado deste ponto?



# Sinais – tentativa 2

```
volatile int count = 0;

void sig_handler(int num) {
    printf("Chamou Ctrl+C\n");
    count++;
}

int main() {
    int count = 0;
    struct sigaction s;
    s.sa_handler = sig_handler;
    sigemptyset(&s.sa_mask);
    s.sa_flags = 0;
    sigaction(SIGINT, &s, NULL);

    if (count == 0)
        printf("Mensagem\n");

    while(1) {
        sleep(1);
    }
    return 0;
}
```

Erro conceitual: O programa principal tenta se sincronizar com o handler

Correto: o handler pode ocorrer a qualquer momento.

# Atividade prática

## Sinais e concorrência (20 minutos)

1. Chamada sigaction e seu uso para receber sinais
2. Sinais diferentes sendo capturados pelo mesmo processo

# Problemas de concorrência!

O que acontece se dois handlers tentam

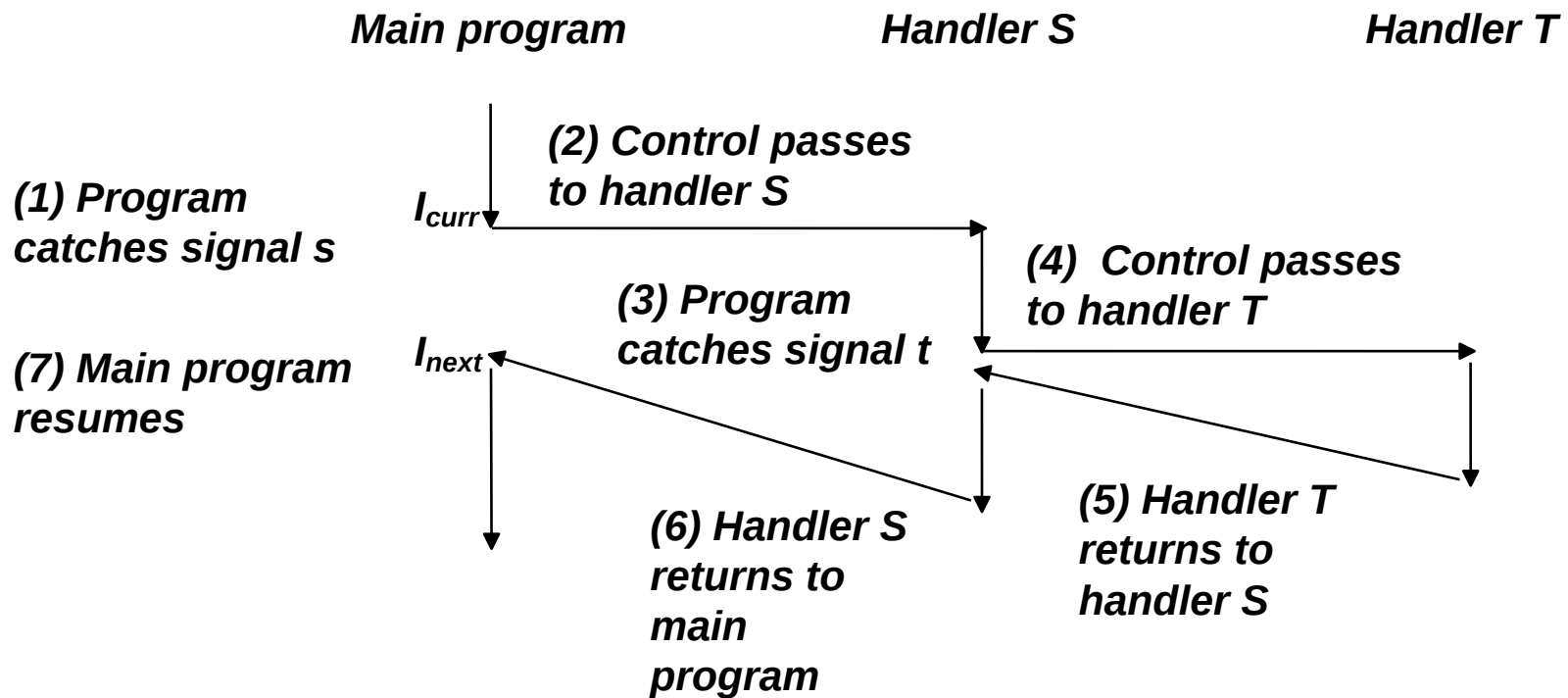
- mexer na mesma variável?
- chamar printf?
- usar a global errno?

Um handler que trata um sinal **A** só pode ser interrompido pela chegada de um outro sinal **B != A**.

Temos que ser cuidadosos ao tratar sinais!

# Handlers aninhados

Handlers podem ser interrompidos por outros handlers!



Mas não pode haver mais de um handler do mesmo sinal rodando!



# Bloqueio de sinais

Podemos "bloquear" o recebimento de um sinal:

- O sinal bloqueado fica pendente até que seja desbloqueado
- Quando for desbloqueado ele será recebido normalmente pelo processo!

**Bloquear um sinal é algo "temporário" e não implica na recepção do sinal**

# Recebendo um sinal

```
struct sigaction {  
    void (*sa_handler)(int);
```

- SIG\_IGN para ignorar
- SIG\_DFL para padrão
- Nome de uma função

```
        void (*sa_sigaction)(int,  
siginfo_t*, void *);
```

```
    sigset_t sa_mask;
```

Sinais a serem bloqueados durante a execução de sa\_handler

```
    int sa_flags;
```

Opções de recepção. Usaremos 0 sempre aqui.

```
        void (*sa_restorer)(void);
```

```
};
```

# Atividade prática

## Bloqueando sinais (15 minutos)

1. Sinais diferentes sendo capturados pelo mesmo processo
2. Bloqueando sinais durante a execução do handler

# Insper

[www.insper.edu.br](http://www.insper.edu.br)