

Universidade Federal do Amazonas  
Faculdade de Tecnologia  
Departamento de Eletrônica e Computação



# *Introdução a Programação Concorrente*

Lucas Cordeiro

[lucascordeiro@ufam.edu.br](mailto:lucascordeiro@ufam.edu.br)

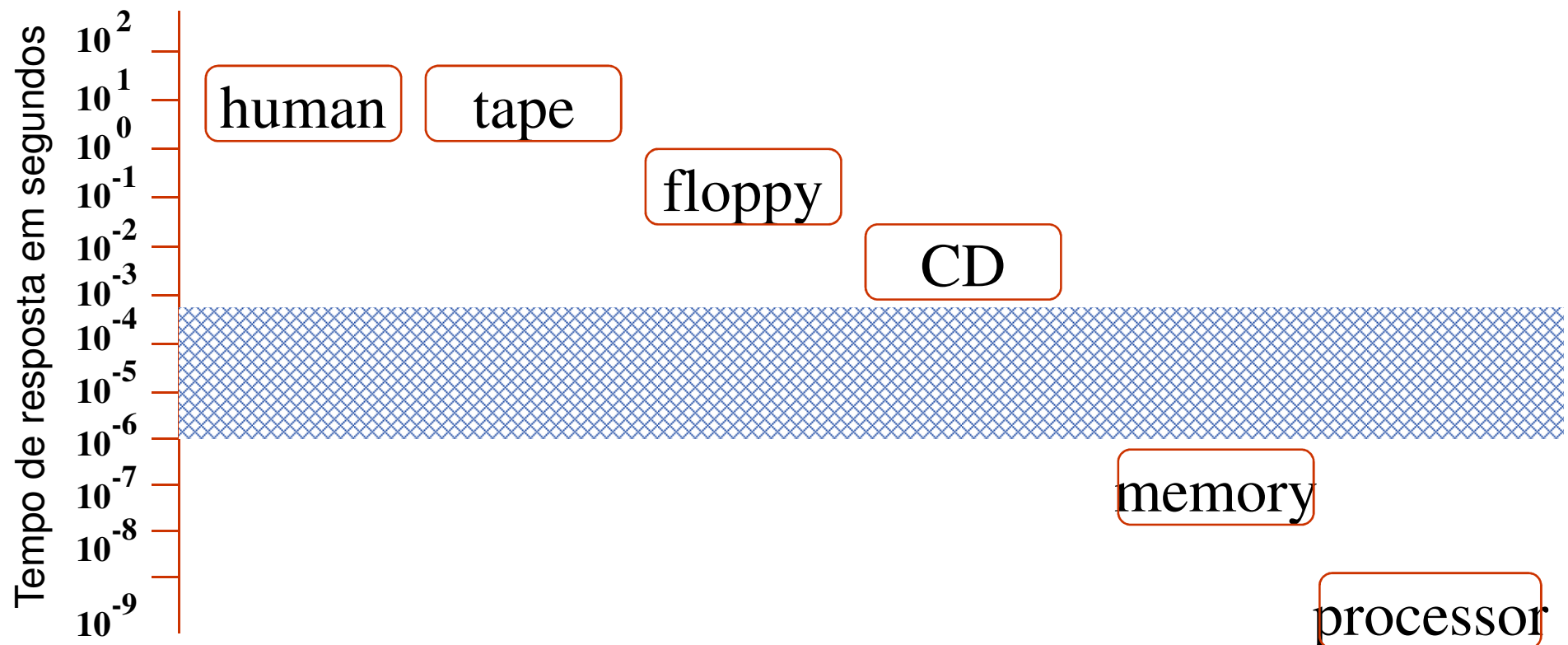
# Programação Concorrente



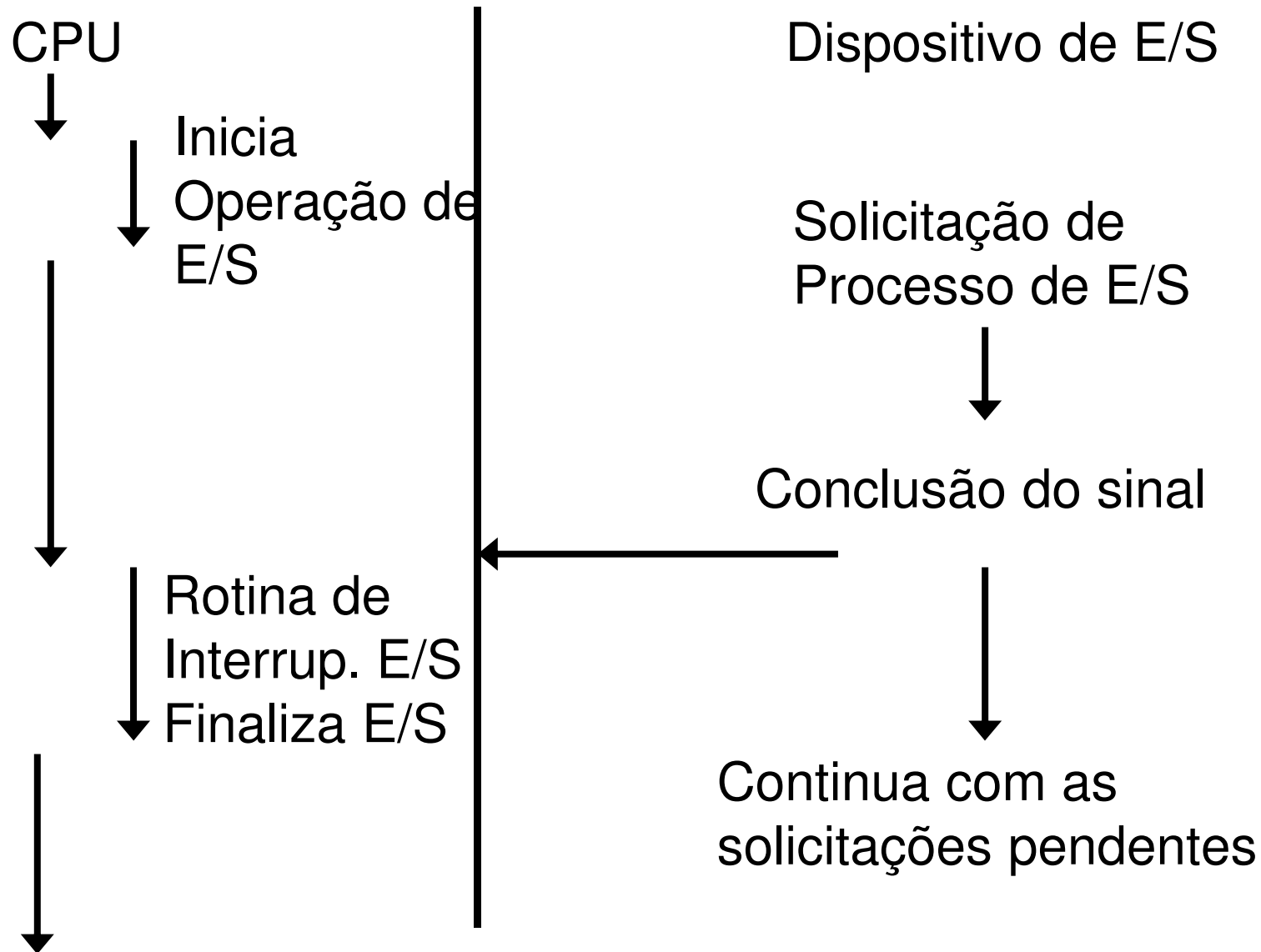
- O nome dado a notação de programação e técnicas para **expressar o paralelismo** e resolver problemas de **comunicação e sincronização**
- **Implementação de paralelismo** é um tópico em sistemas de computação (hardware e software) que é essencialmente **independente de programação concorrente**
- **Programação concorrente** é importante porque proporciona um **ambiente abstrato** para estudar o paralelismo sem se “atolar” em detalhes de implementação do hardware

# Por quê nós precisamos disso?

- Para nós podermos utilizar completamente o processador



# Paralelismo entre CPU e Dispositivos de E/S



# Por quê nós precisamos disso?



- Permitir a expressão do potencial paralelismo para que **mais de um computador** possa ser usado para resolver o problema
- Considere a tentativa de encontrar o caminho através de um labirinto





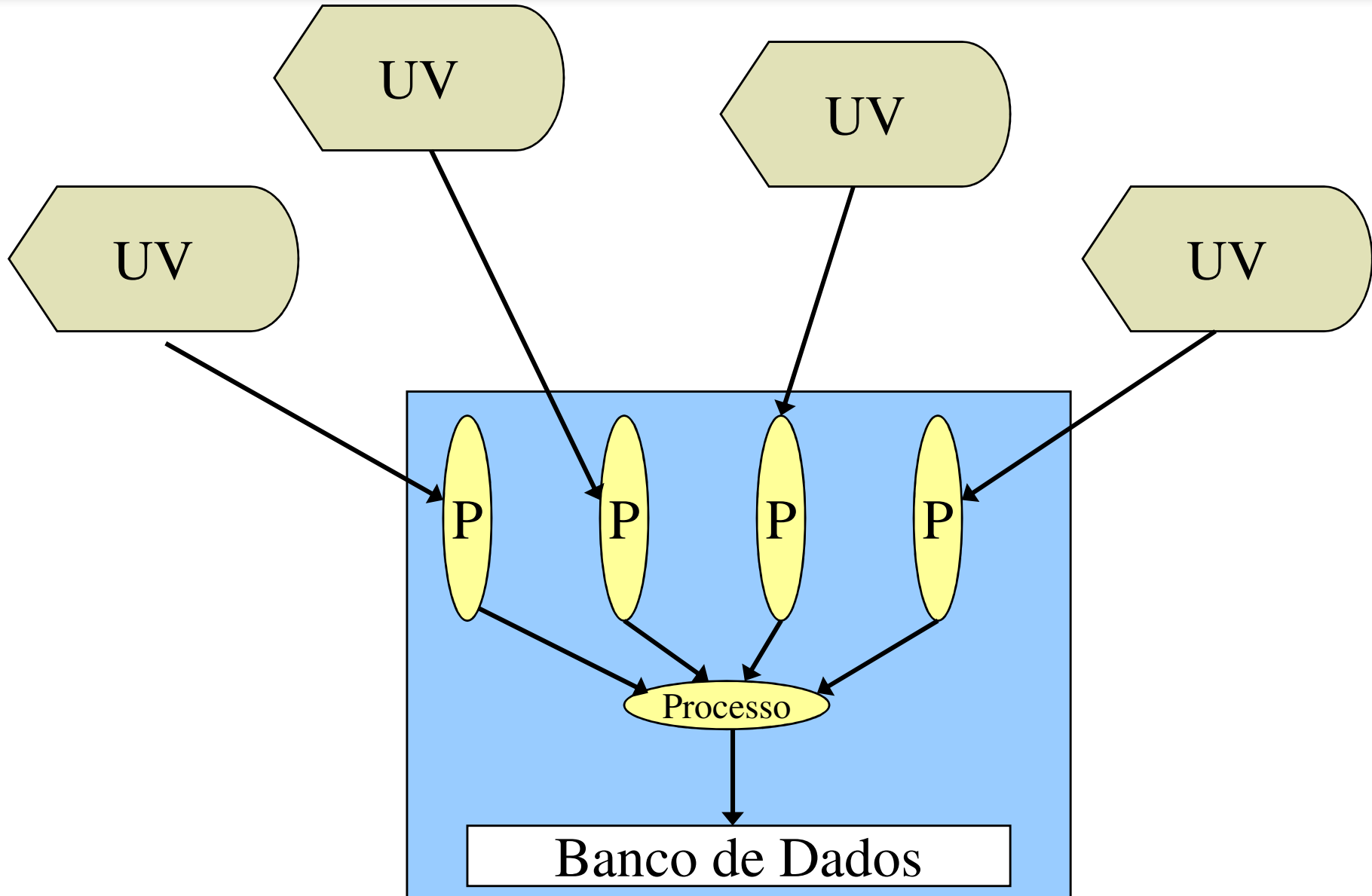
# Por quê nós precisamos disso?



- Para modelar o paralelismo no mundo real
- Praticamente todos os **sistemas de tempo real** são **inerentemente concorrentes** — dispositivos operam em paralelo no mundo real
- Este é, talvez, a principal razão para usar a concorrência



# Sistema de reservas aéreas



# Por quê nós precisamos disso?

- A alternativa é usar técnicas de **programação seqüencial**
- O programador deve construir o sistema tal que envolva a **execução cíclica** para lidar com as atividades concorrentes
  - isto complica a difícil tarefa do programador e o envolve em considerações de estruturas que são irrelevantes para o controle
- Os programas resultantes serão mais **obscuros e deselegantes**
- Dificulta a **decomposição** de problemas mais complexos
- Execução paralela do programa **em mais de um processador** será muito mais difícil de alcançar
- A inserção de código para lidar com **falhas** é mais problemático

# Terminologia

- Um programa concorrente é uma **coleção de processos seqüenciais autônomos**, executado (logicamente) em paralelo
- Cada processo tem uma única *thread* de controle
- A implementação real (*i.e.*, execução) de uma coleção de processos geralmente exhibe uma das três formas

## Multiprogramming

- processos multiplicam suas execuções em um único processador

## Multiprocessing

- processos multiplicam suas execuções em um sistema multi-processador onde existe acesso a memória compartilhada

## Distributed Processing

- processos multiplicam suas execuções em vários processadores que não compartilham memória

# Processos e *Threads* (1)

- Todos os sistemas operacionais fornecem **processos**
- **Processos** executam nas suas próprias **máquinas virtuais** (VM) para evitar interferência de outros processos
- Recentes SOs fornecem mecanismos para criar **threads** dentro da mesma VM
  - threads são as vezes fornecidas de forma transparente para o SO
- Threads têm acesso irrestrito a suas VM
- Um simples processo pode ter **múltiplas threads** que **compartilham dados** globais e espaço de endereços
  - Threads podem operar no mesmo conjunto de dados facilmente
- **Processos não compartilham** espaço de endereço

# Processos e *Threads* (2)



- O programador e a linguagem devem fornecer a proteção de interferências
- Longo debate se a linguagem deve definir concorrência ou deixar para o SO
  - Ada and Java fornecem concorrência
  - C, C++ não

# Como Visualizar Processos no Linux?

- o seguinte comando visualiza todos os processos que estão executando no PC:
  - `$ps aux | less`
- o comando `top` fornece uma visão em tempo real dinâmica dos processos que estão sendo executados:
  - `$top`

```
File Edit View Terminal Help
^top - 03:28:25 up 5:08, 4 users, load average: 0.03, 0.05, 0.01
Tasks: 218 total, 2 running, 216 sleeping, 0 stopped, 0 zombie
Cpu0  : 0.7%us, 1.3%sy, 0.0%ni, 98.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1  : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2  : 0.0%us, 0.0%sy, 0.0%ni, 100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3  : 0.7%us, 0.0%sy, 0.0%ni, 99.3%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem:   8193284k total, 2054268k used, 6139016k free, 184712k buffers
Swap:  0k total,      0k used,      0k free, 1099976k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
  7287 vivek    20   0  715m 175m 28m  S   1  2.2   1:55.96  firefox
  7485 vivek    20   0  212m  16m 10m  S   1  0.2    0:02.33  gnome-terminal
    75 root     15  -5    0    0    0  S   0  0.0    0:02.66  scsi_eh_4
  1520 root     20   0 22180 1240 1040  S   0  0.0    0:05.19  hald-addon-stor
  1554 root     20   0  387m  47m  15m  S   0  0.6    9:44.74  Xorg
  7352 vivek    20   0  146m  31m  11m  S   0  0.4    0:16.51  npviewer.bin
    1 root     20   0 19456 1880 1204  S   0  0.0    0:01.01  init
    2 root     15  -5    0    0    0  S   0  0.0    0:00.00  kthreadd
    3 root     RT  -5    0    0    0  S   0  0.0    0:00.01  migration/0
    4 root     15  -5    0    0    0  S   0  0.0    0:00.09  ksoftirqd/0
    5 root     RT  -5    0    0    0  S   0  0.0    0:00.00  watchdog/0
    6 root     RT  -5    0    0    0  S   0  0.0    0:00.00  migration/1
    7 root     15  -5    0    0    0  S   0  0.0    0:00.03  ksoftirqd/1
    8 root     RT  -5    0    0    0  S   0  0.0    0:00.00  watchdog/1
```

# Representação de Processos



- *fork e join*
- *cobegin*
- declaração de processo explícita

# Fork e Join

- O ***fork*** especifica que uma designada rotina deve iniciar sua **execução concorrentemente** com o ***invoker***
- ***join*** permite que o ***invoker*** **aguarde pelo término** da rotina invocada

```
function F return is ...;
procedure P;
    ...
    C := fork F;
    ...
    J := join C;
    ...
end P;
```

- Depois do ***fork***, ***P*** e ***F*** executarão concorrentemente. No ponto do ***join***, ***P*** aguardará até que ***F*** tenha terminado (se não o tiver feito)
- A notação do ***fork*** e ***join*** podem ser encontradas no UNIX/POSIX



# Exemplo do UNIX Fork

```
for (I=0; I!=10; I++) {  
    pid[I] = fork();  
}  
wait . . .
```

Quantos processos são criados?

# Cobegin

- *cobegin* (ou *parbegin* ou *par*) é uma maneira estruturada de denotar a execução concorrente de uma coleção de comandos:

*cobegin*

$S_1$ ;

$S_2$ ;

$S_3$ ;

...

$S_n$

*coend*

- $S_1$ ,  $S_2$ , etc, executam concorrentemente
- Os comandos terminam quando  $S_1$ ,  $S_2$ , etc terminarem
- Cada  $S_i$  pode ser uma instrução permitida da linguagem
- *cobegin* pode ser encontrado em *Edison* e *occam2*

# Declaração de Processo Explícita

- A estrutura de um programa pode ser feita de forma mais clara se as declarações das rotinas indicam se elas serão executadas concorrentemente
- Note que isto não quer dizer quando elas executarão

```
task body Process is  
begin  
    ...  
end;
```

- Linguagens que suportam declaração de processo explícita podem ter **criação de processo/tarefa implícita ou explícita**

# Tasks e Ada

- A unidade de concorrência em Ada é conhecida como **task**
- Tarefas devem ser explicitamente declaradas, não existe comandos *fork/join*, *COBEGIN/PAR*, etc
- Tarefas podem ser declaradas em qualquer nível do programa
  - elas são criadas implicitamente uma vez que entram no escopo das suas declarações via a ação de um coletor
- Tarefas podem comunicar e sincronizar através de uma variedade de mecanismos : **rendezvous** (uma forma de passagem de mensagem sincronizada), **unidades protegidas** (uma forma de região crítica de monitor/condicional), e **variáveis compartilhadas**

# Exemplo de Estrutura de Tarefa

```
task type Server (Init : Parameter) is  
    entry Service;  
end Server;
```

specification

```
task body Server is  
begin  
    ...  
    accept Service do  
        -- Sequence of statements;  
    end Service;  
    ...  
end Server;
```

body

# Exemplo de Especificações de Tarefas

```
task type Controller;
```

este tipo de tarefa não tem *entries*; nenhuma outra tarefa pode se comunicar diretamente

```
task type Agent(Param : Integer);
```

este tipo de tarefa não tem *entries* mas objetos de tarefa podem ser passados (um parâmetro inteiro) nos seus tempos de criação

```
task type Garage_Attendant(  
    Pump : Pump_Number := 1) is  
    entry Serve_Leaded(G : Gallons);  
    entry Serve_Unleaded(G : Gallons);  
end Garage_Attendant;
```

objetos permitirão comunicação através de duas *entries*; o número da bomba a ser servida é passada na criação da tarefa; se nenhuma válvula é passada um valor *default* de 1 é usado

# Um Procedimento com duas Tarefas

```
procedure Example1 is
  task A;
  task B;

  task body A is
    -- local declarations for task A
  begin
    -- sequence of statement for task A
  end A;

  task body B is
    -- local declarations for task B
  begin
    -- sequence of statements for task B
  end B;
begin
  -- tasks A and B start their executions before
  -- the first statement of the procedure's sequence
  -- of statements.
  ...
end Example1; -- the procedure does not terminate
               -- until tasks A and B have
               -- terminated.
```

# Executando o Exemplo com duas Tarefas

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Example1 is
  task A;
  task B;

  task body A is
  begin
    Put_Line("Task A...");
  end A;

  task body B is
  begin
    Put_Line("Task B...");
  end B;

begin
  Put_Line("Main Procedure");
end Example1;
```

```
lucas@Lucas ada95]$ ./example1
Task B...
Main Procedure
Task A...
[lucas@Lucas ada95]$
./example1
Task A...
Task B...
Main Procedure
[lucas@Lucas ada95]$
./example1
Task B...
Task A...
Main Procedure
[lucas@Lucas ada95]$
./example1
Task A...
Main Procedure
Task B...
```



# Exemplo com Rendezvous

## (passagem de mensagem)

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure example2 is
  task A is entry pointA(M: in Integer); end A;
  task B is entry pointB(M: in Integer); end B;
  task body A is
  begin
    accept pointA(M: in Integer) do
      Put_Line("Starting task A...");
      Put("The number in Task A is "); Put(M); New_line;
    end pointA;
  end A;
  task body B is
  begin
    accept pointB(M: in Integer) do
      Put_Line("Starting task B...");
      Put("The number in Task B is "); Put(M); New_line;
    end pointB;
  end B;
```

# Exemplo com Rendezvous

(passagem de mensagem)

**begin**

```
Put_Line("Main Procedure");
```

```
A.pointA(1);
```

```
B.pointB(2);
```

**end** example2;

```
[lucas@Lucas ada95]$ ./example2
Main Procedure
Starting task A...
The number in Task A is          1
Starting task B...
The number in Task B is          2
[lucas@Lucas ada95]$ ./example2
Main Procedure
Starting task A...
The number in Task A is          1
Starting task B...
The number in Task B is          2
```

# Criação Dinâmica de Tarefa

- Atribuindo valores *non-static* para os limites de um vetor (de tarefas), um número dinâmico de tarefas é criado
- Criação de tarefas dinâmicas podem ser obtidas explicitamente usando o operador “**new**” em um tipo de acesso (de um tipo de tarefa)

```
procedure Example2 is
```

```
  task type T;
```

```
  type A is access T;
```

```
  P : A;
```

```
  Q : A := new T;
```

```
  ...
```

```
begin
```

```
  ...
```

```
  P := new T;
```

```
  Q := new T;  -- O que acontece ao antigo Q?
```

```
  ...
```

```
end example2;
```

Isto cria uma tarefa que imediatamente começa a sua inicialização e execução;

# Ativação, Execução & Finalização



A execução de um objeto de tarefa tem três principais fases:

- *Ativação* — a elaboração da parte declarativa, se houver, do corpo de tarefas (todas as **variáveis locais** da tarefa são **criadas e inicializadas** durante esta fase)
- *Execução Normal* — a **execução dos comandos** dentro do corpo de uma tarefa
- *Finalização* — a **execução** de qualquer **código de finalização** associado com todos os objetos em sua parte declarativa

# Ativação de Tarefa

**declare**

**task type** T\_Type1;

Criado quando a declaração é elaborada

**task** A;

B, C : T\_Type1;

B e C criado quando elaborado

**task body** A **is** ...;

**task body** T\_Type1 **is**

...

**begin**

tarefas ativadas quando a elaboração é finalizada

...

**end;**

Primeiro comando executa uma vez que todas as tarefas tenham terminado as suas ativações

# Sincronização e Comunicação

- O **comportamento correto** de um programa concorrente depende da **sincronização** e **comunicação** entre os processos
- *Sincronização*: a satisfação das **restrições na intercalação** das ações dos processos (p.e., um ação de processo ocorrendo somente depois de uma ação de um outro processo)
- *Comunicação*: **passagem de informações** de um processo para o outro
  - conceitos estão entrelaçados pois comunicação solicita sincronização e sincronização pode ser considerado como comunicação sem conteúdo
  - **comunicação** de dados é geralmente baseada em **variáveis compartilhadas** e **passagem de mensagens**

# Comunicação de Variável Compartilhada

- Exemplos: *espera ocupante, semáforos e mutexes*
- Uso irrestrito de **variáveis compartilhadas** não é confiável e também inseguro devido a **problemas de atualizações múltiplas**
- Considere dois processos atualizando uma variável compartilhada,  $X$ , com a atribuição :  $X := X + 1$ 
  - carrega o valor de  $X$  em algum registrador
  - incrementa o valor no registrador por 1
  - armazena o valor no registrador de volta para  $X$
- Como as três operações não são indivisíveis, dois processos simultaneamente atualizando a variável podem seguir uma linha de execução que produziria **resultados incorretos**

```
movl x, $t0  
addl $1, $t0  
movl $t0, x
```

# Microbenchmark [Ghafari et al. 2010]

```
#include <assert.h>
#include <pthread.h>
int x=0;
void* tr(void* arg) {
    x++;
    x++;
    x++;
    x++;
    x++;
    x++;
    assert(x<=6);
}
int main(void) {
    pthread_t id[4];
    pthread_create(&id[0], NULL, &tr, NULL);
    pthread_create(&id[1], NULL, &tr, NULL);
    pthread_create(&id[2], NULL, &tr, NULL);
    pthread_create(&id[3], NULL, &tr, NULL);
    return 0;
}
```

Possível contraexemplo:

tr thread1 x=1	tr thread2 x=7
tr thread1 x=2	tr thread2 x=8
tr thread1 x=3	tr thread2 x=9
tr thread1 x=4	tr thread2 x=10
tr thread1 x=5	tr thread2 x=11
tr thread1 x=6	tr thread2 x=12



# Comunicação de recurso compartilhado

```
type Coordinates is  
  record  
    X : Integer;  
    Y : Integer;  
  end record;  
Shared_Cordinate: Coordinates;
```

```
task body Helicopter is  
  Next: Coordinates;  
begin  
  loop  
  
  Compute_New_Cordinates (Next);  
  Shared_Cordinates := Next;  
  
  end loop  
end;
```

```
task body Police_Car is  
begin  
  loop  
  
  Plot (Shared_Cordinates);  
  
  end loop;  
end;
```

$X = 5$

$Y = 4$

5

4

1,1  
2,2  
3,3  
4,4  
5,5  
6,6  
...

Rota de fuga do vilão visto por um helicóptero

1,1  
2,2  
3,3  
4,4  
5,4

Carro da polícia  
rota de busca

**Vilão**  
**Escapa!**

# Evitando Interferências



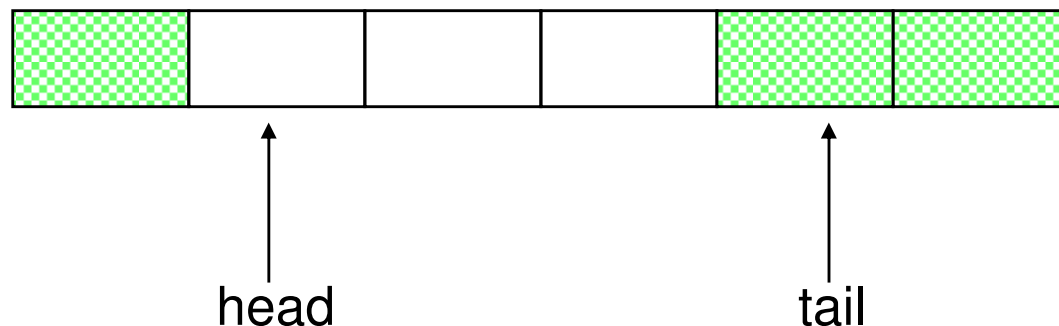
- As partes de um processo que acessam variáveis compartilhadas devem ser executadas indivisivelmente com relação ao outro
- Estas partes são chamadas **seções críticas**
- A proteção necessária é chamada **exclusão mútua**

# Exclusão Mútua

- A **seqüência de instruções** que deve ser **executada indivisivelmente** é chamado de **seção crítica**
- A sincronização necessária para proteger uma seção crítica é conhecida como **exclusão mútua**
- **Atomicidade** é assumida para estar presente no nível de memória. Se um processo está executando  $X := 5$ , simultaneamente com outro executando  $X := 6$ , o resultado será ou 5 ou 6 (não outro valor)
- Se dois processos estão atualizando um objeto estruturado, esta **atomicidade** somente será aplicada no **nível de elemento**

# Sincronização de Condição

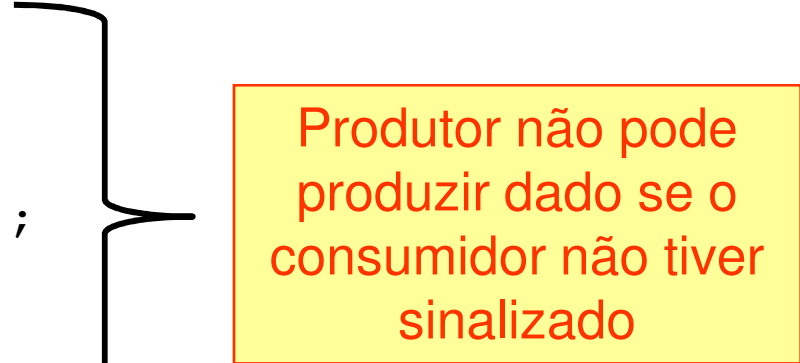
- Sincronização de condição é necessária quando um processo deseja executar **uma operação** que só pode ser realizado com segurança se **outro processo** tenha tomado alguma ação ou está em algum **estado definido**
- P.e. um *buffer* limitado tem 2 condições de sincronização:
  - os **processos do produtor** não devem tentar depositar dados no *buffer* se o *buffer* estiver cheio
  - os **processos do consumidor** não são permitidos extrair objetos do *buffer* se o *buffer* estiver vazio



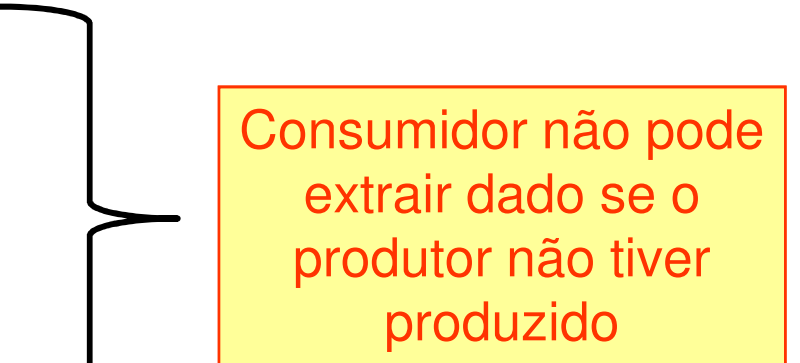
Exclusão  
mútua é  
necessário?

# Exemplo: Produtor/Consumidor (1)

```
#include <stdio.h>
#include <pthread.h>
int num; pthread_mutex_t m;
pthread_cond_t empty, full;
void *producer(void *arg) {
    while(1) {
        pthread_mutex_lock(&m);
        while (num > 0)
            pthread_cond_wait(&empty, &m);
        num++;
        pthread_mutex_unlock(&m);
        pthread_cond_signal(&full);
    }
}
void *consumer(void *arg) {
    while(1) {
        pthread_mutex_lock(&m);
        while (num == 0)
            pthread_cond_wait(&full, &m);
        num--;
```



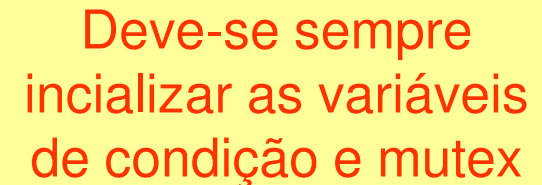
Produtor não pode produzir dado se o consumidor não tiver sinalizado



Consumidor não pode extrair dado se o produtor não tiver produzido

# Exemplo: Produtor/Consumidor (2)

```
    pthread_mutex_unlock(&m);  
    pthread_cond_signal(&empty);  
}  
}  
int main(void) {  
    pthread_t  t1, t2;  
    num = 1;  
    pthread_mutex_init(&m, 0);  
    pthread_cond_init(&empty, 0);  
    pthread_cond_init(&full, 0);  
    pthread_create(&t1, 0, producer, 0);  
    pthread_create(&t2, 0, consumer, 0);  
    pthread_join(t1, 0);  
    pthread_join(t2, 0);  
    return 0;  
}
```



Deve-se sempre  
inicializar as variáveis  
de condição e mutex

# Espera Ocupante



- Uma maneira de implementar a sincronização é ter processos definidos e verificar as **variáveis compartilhadas** que estão atuando como **flags**
- Algoritmos de espera ocupante são em geral **ineficientes**; eles envolvem processos consumindo todos os ciclos de processamento quando eles não conseguem executar um trabalho proveitoso
- Mesmo em um sistema multi-processador eles podem dar origem a **tráfego excessivo** no barramento de memória ou rede (se distribuído)



# Exemplo de Espera Ocupante

- Para sinalizar uma condição, uma tarefa seta o valor de uma *flag*
  - Para esperar por esta condição, uma outra tarefa checa esta *flag* e procede somente quando o valor apropriado for lido

```
task P1;  -- pseudo code for waiting task
```

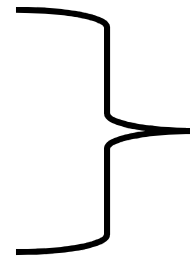
```
...  
  while flag = down do  
    null  
  end;
```

```
...  
end P1;
```

```
task P2 -- signalling task
```

```
...  
  flag := up;
```

```
...  
end P2;
```



Fica no laço e  
checando a flag caso  
a mesma não tenha  
sido setada

*Busy waiting ou spinning  
flag chamada de spin lock*

# Semáforos

- Um semáforo é uma **variável inteira não negativa** que, além da inicialização só pode ser postas em prática por dois procedimentos: P (ou WAIT) e V (ou SIGNAL)
- **WAIT(S)** se o valor de  $S > 0$  então decrementa seu valor por um; de outro modo retarda o processo até que  $S > 0$  (e então decrementa o seu valor)
- **SIGNAL(S)** incrementa o valor de S por um
- WAIT e SIGNAL são atômicos (indivisíveis). Dois processos ambos executando **operações WAIT** no mesmo semáforo **não podem interferir** um ao outro e **não podem falhar** durante a execução de uma operação de semáforo

# Sincronização de Condição

```
var consyn : semaphore (* init 0 *)
```

```
process P1;  
  (* waiting process *)  
  statement X;  
  wait (consyn)  
  statement Y;  
end P1;
```

```
process P2;  
  (* signalling proc *)  
  statement A;  
  signal (consyn)  
  statement B;  
end P2;
```

Em qual ordem as instruções executarão?

# Exclusão Mútua

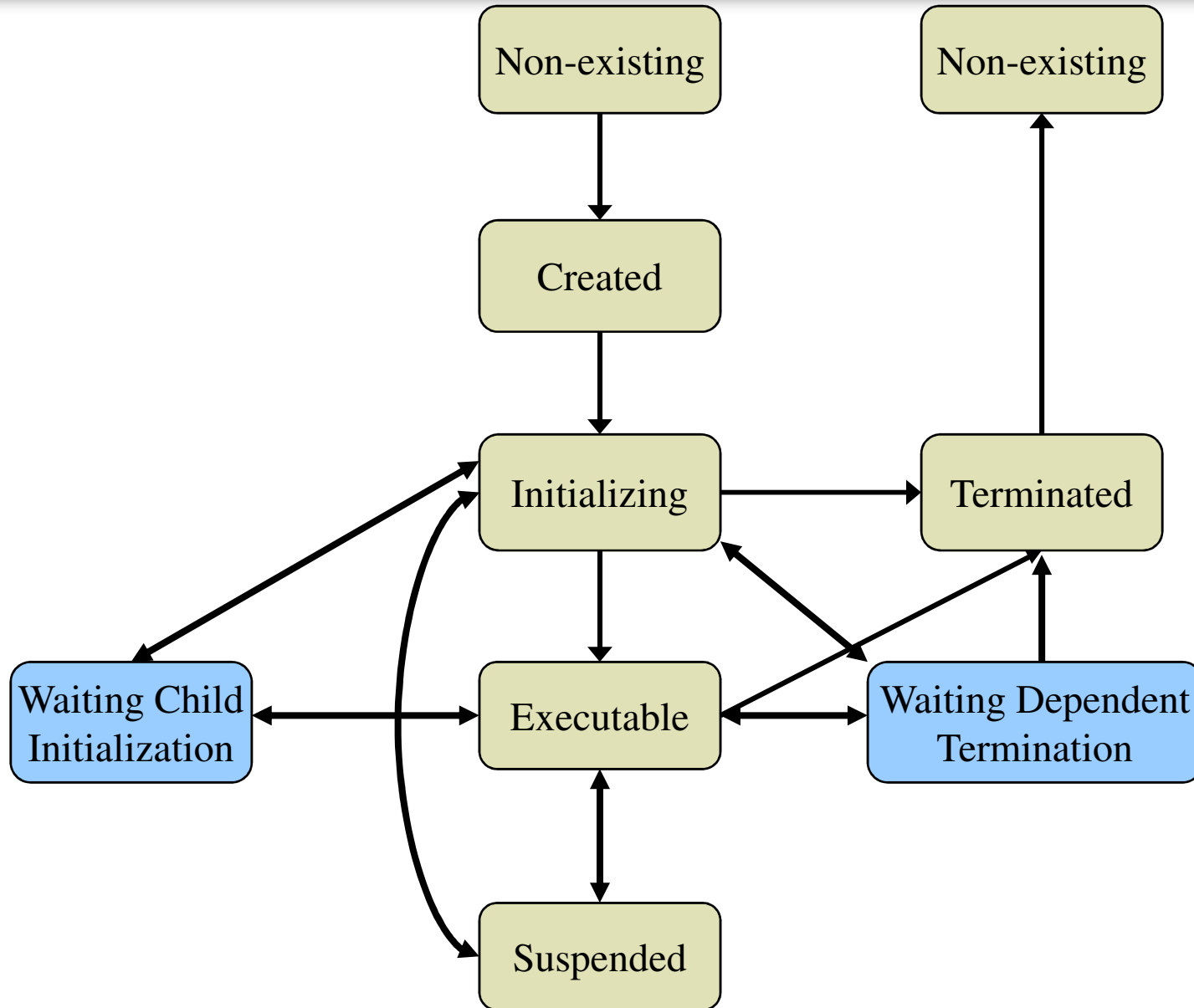
```
(* mutual exclusion *)  
var mutex : semaphore; (* initially 1 *)
```

```
process P1;  
  statement X  
  wait (mutex);  
  statement Y  
  signal (mutex);  
  statement Z  
end P1;
```

```
process P2;  
  statement A;  
  wait (mutex);  
  statement B;  
  signal (mutex);  
  statement C;  
end P2;
```

Em qual ordem as instruções executarão?

# Estados do Processo



# Deadlock

- Dois processos estão num impasse (*deadlock*) se cada um está segurando um recurso enquanto espera por um recurso mantido por outro

```
type Sem is ...;  
X : Sem := 1; Y : Sem := 1;
```

```
task A;  
task body A is  
begin  
    ...  
    Wait(X);  
    Wait(Y);  
    ...  
end A;
```

```
task B;  
task body B is  
begin  
    ...  
    Wait(Y);  
    Wait(X);  
    ...  
end B;
```

# Livelock

- Dois processos estão no estado *livelock* se cada um está em execução mas não é capaz de fazer progresso

```
type Flag is (Up, Down);  
Flag1 : Flag := Up;
```

```
task A;  
task body A is  
begin  
  ...  
  while Flag1 = Up loop  
    null;  
  end loop;  
  ...  
end A;
```

```
task B;  
task body B is  
begin  
  ...  
  while Flag1 = Up loop  
    null;  
  end loop;  
  ...  
end A;
```

# Semáforos binários e quantidade

- Um **semáforo geral** é um inteiro não negativo; seu valor pode subir para qualquer número positivo suportado
- Um **semáforo binário** somente assume o valor 0 e 1; a sinalização de um semáforo que tem o valor 1 não tem efeito – o semáforo retém o valor 1
- Com um **semáforo de quantidade** a quantidade a ser decrementada pelo WAIT (e incrementada pelo SIGNAL) é dada como um parâmetro; por exemplo, *WAIT (S, i)*

```
wait(S, i)  :- if S >= i then
                S := S - i;
            else
                delay
                S := S - i
signal(S, i) :- S := S+i
```



# Programas com Semáforo em Ada

```
package Semaphore_Package is  
  type Semaphore(Initial : Natural) is limited private;  
  procedure Wait (S : Semaphore);  
  procedure signal (S : Semaphore);  
private  
  type Semaphore ...  
end Semaphore_Package;
```

- Ada não suporta diretamente semáforos; os procedimentos de *wait* e *signal* podem, porém, serem construídos a partir das primitivas de sincronização do Ada
- A essência dos tipos de dados abstrato é que eles podem ser usado sem o conhecimento das suas implementações!

# O Buffer Limitado

```
package Buffer is
  procedure Append (I : Integer);
  procedure Take (I : out Integer);
end Buffer;

package body Buffer is
  Size : constant Natural := 32;
  type Buffer_Range is mod Size;
  Buf : array (Buffer_Range) of Integer;
  Top, Base : Buffer_Range := 0;
  Mutex : Semaphore(1);
  Item_Available : Semaphore(0);
  Space_Available : Semaphore(Size);
  procedure Append (I : Integer) is separate;
  procedure Take (I : out Integer) is separate;
end Buffer;
```

# O Buffer Limitado

```
procedure Append(I : Integer) is  
begin  
    Wait(Space_Available);  
    Wait(Mutex);  
    Buf(Top) := I;  
    Top := Top+1  
    Signal(Mutex);  
    Signal(Item_Available);  
end Append;
```

```
procedure Take(I : out Integer) is  
begin  
    Wait(Item_Available);  
    Wait(Mutex);  
    I := BUF(base);  
    Base := Base+1;  
    Signal(Mutex);  
    Signal(Space_Available);  
end Take;
```

# POSIX Mutexes e Variáveis de Condição

- Fornece sincronização entre as tarefas
- Mutexes e variáveis de condição têm objetos de atributos associados; nós usaremos os atributos padrões somente
- Exemplo de atributos:
  - Definir a semântica para uma tarefa tentando dar um *lock* num mutex que já está em *lock*
  - Permitir compartilhamento de mutexes e variáveis de condição entre processos
  - definir/obter a prioridade teto
  - definir/obter o *clock* usado para *timeouts*

```
typedef ... pthread_mutex_t;  
typedef ... pthread_mutexattr_t;  
typedef ... pthread_cond_t;  
typedef ... pthread_condattr_t;
```

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr);
/* initialises a mutex with certain attributes */

int pthread_mutex_destroy(pthread_mutex_t *mutex);
/* destroys a mutex */
/* undefined behaviour if the mutex is locked */

int pthread_cond_init(pthread_cond_t *cond,
                     const pthread_condattr_t *attr);
/* initialises a condition variable with certain attributes */

int pthread_cond_destroy(pthread_cond_t *cond);
/* destroys a condition variable */
/* undefined, if threads are waiting on the cond. variable */
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
    /* lock the mutex; if locked already suspend calling thread */
    /* the owner of the mutex is the thread which locked it */

int pthread_mutex_trylock(pthread_mutex_t *mutex);
    /* as lock but gives an error if mutex is already locked */

int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                             const struct timespec *abstime);
    /* as lock but gives an error if mutex cannot be obtained */
    /* by the timeout */

int pthread_mutex_unlock(pthread_mutex_t *mutex);
    /* unlocks the mutex if called by the owning thread */
    /* undefined behaviour if calling thread is not the owner */
    /* undefined behaviour if the mutex is not locked */
    /* when successful, a blocked thread is released */
```

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
    /* called by thread which owns a locked mutex */
    /* undefined behaviour if the mutex is not locked */
    /* atomically blocks the caller on the cond variable and */
    /* releases the lock on mutex */
    /* a successful return indicates the mutex has been locked */

int pthread_cond_timedwait(pthread_cond_t *cond,
                            pthread_mutex_t *mutex, const struct timespec *abstime);
    /* the same as pthread_cond_wait, except that a error is */
    /* returned if the timeout expires */
```

```
int pthread_cond_signal(pthread_cond_t *cond);
    /* unblocks at least one blocked thread */
    /* no effect if no threads are blocked */

int pthread_cond_broadcast(pthread_cond_t *cond);
    /* unblocks all blocked threads */
    /* no effect if no threads are blocked */

/*all unblocked threads automatically contend for */
/* the associated mutex */
```

**Todas as funções retornam 0 se forem executadas com sucesso**



# Buffer Limitado no POSIX

```
#define BUFF_SIZE 10

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t buffer_not_full;
    pthread_cond_t buffer_not_empty;
    int count, first, last;
    int buf[BUFF_SIZE];
} buffer;

int append(int item, buffer *B ) {
    PTHREAD_MUTEX_LOCK(&B->mutex);
    while(B->count == BUFF_SIZE) {
        PTHREAD_COND_WAIT(&B->buffer_not_full, &B->mutex); }
    /* put data in the buffer and update count and last */
    PTHREAD_MUTEX_UNLOCK(&B->mutex);
    PTHREAD_COND_SIGNAL(&B->buffer_not_empty);
    return 0;
}
```

```
int take(int *item, buffer *B ) {
    pthread_mutex_lock(&B->mutex);
    while(B->count == 0) {
        pthread_cond_wait(&B->buffer_not_empty, &B->mutex);
    }
    /* get data from the buffer and update count and first */
    pthread_mutex_unlock(&B->mutex);
    pthread_cond_signal(&B->buffer_not_full);
    return 0;
}
```

```
int initialize(buffer *B) {
    /* set the attribute objects and initialize the */
    /* mutexes and condition variable */
}
```

# Controlador do Braço de um Robô (1)

```
#include <pthread.h>
#include <stdlib.h>

pthread_attr_t attributes;
pthread_t xp, yp, zp;
typedef enum {xplane, yplane, zplane} dimension;
int new_setting(dimension D);
void move_arm(dimension D, int P);
void controller(dimension *dim) {
    int position, setting;
    position = 0;
    while(1) {
        move_arm(*dim, position);
        setting = new_setting(*dim);
        position = position + setting;
    }
}
```

# Controlador do Braço de um Robô (2)

```
int main() {
    dimension X, Y, Z;
    void *result;
    X = xplane;
    Y = yplane;
    Z = zplane;
    if (pthread_attr_init(&attributes) != 0)
        exit(EXIT_FAILURE);
    if (pthread_create(&xp, &attributes, (void *)controller, &X) != 0)
        exit(EXIT_FAILURE);
    if (pthread_create(&yp, &attributes, (void *)controller, &Y) != 0)
        exit(EXIT_FAILURE);
    if (pthread_create(&zp, &attributes, (void *)controller, &Z) != 0)
        exit(EXIT_FAILURE);
    pthread_join(xp, (void **)&result);
    exit(EXIT_FAILURE);
}
```

# Concorrência em Java

- Java tem uma classe pré-definida `java.lang.Thread` que fornece mecanismos através dos quais *threads* são criadas
- Para evitar que todas as threads tenham que ser filhas de `Thread`, Java tem uma interface padrão

```
public interface Runnable {  
    public abstract void run();  
}
```

- Qualquer classe que deseje expressar execução de concorrência deve implementar esta interface e fornecer o método `run`

# A Classe Thread

```
public class Thread extends Object implements Runnable
{
    public Thread(); // Allocates a new Thread object
    public Thread(Runnable target);
    public void run();
    public void start();
    public static Thread currentThread();
    public final void join() throws InterruptedException;
    public final boolean isAlive(); //A thread is alive if it has
                                    // been started and has not yet
                                    // died

    public final void setDaemon(); // Marks this thread as
    public final boolean isDaemon(); // either a daemon thread or
                                    // a user thread
}
```

Uma thread *daemon* é terminada pela JVM quando nenhuma outra thread (incluindo a *main*) esteja executando (*garbage collector*)

# Braço do Robô: Subclasse de *Thread* (1)

```
public class UserInterface
{
    public int newSetting (int Dim) { ... }
    ...
}
```

Classes e  
objetos  
disponíveis

```
public class Arm
{
    public void move(int dim, int pos) { ... }
}
```

```
UserInterface UI = new UserInterface();
```

```
Arm Robot = new Arm();
```

# Braço do Robô: Subclasse de *Thread* (2)

```
public class Control extends Thread {
    private int dim;

    public Control(int Dimension) { // constructor
        super();
        dim = Dimension;
    }
    public void run() {
        int position = 0;
        int setting;

        while(true){
            Robot.move(dim, position);
            setting = UI.newSetting(dim);
            position = position + setting;
        }
    }
}
```

Sobescreve o  
método *run*

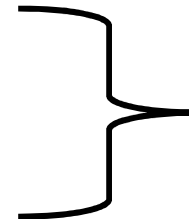
Classe para  
representar os  
três  
controladores



# Braço do Robô: Subclasse de *Thread* (3)

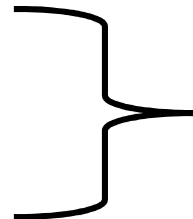
```
final int xPlane = 0; // final indicates a constant  
final int yPlane = 1;  
final int zPlane = 2;
```

```
Control C1 = new Control(xPlane);  
Control C2 = new Control(yPlane);  
Control C3 = new Control(zPlane);
```



Cria as  
*threads*

```
C1.start();  
C2.start();  
C3.start();
```



Inicia as threads

Caso o método `run` seja chamado explicitamente, então o código executará sequencialmente

# Braço do Robô: Implementa *Runnable* (1)

```
public class Control implements Runnable {
    private int dim;

    public Control(int Dimension) { // constructor
        dim = Dimension;
    }

    public void run() {
        int position = 0;
        int setting;

        while(true) {
            Robot.move(dim, position);
            setting = UI.newSetting(dim);
            position = position + setting;
        }
    }
}
```

# Braço do Robô: Implementa *Runnable* (2)

```
final int xPlane = 0;  
final int yPlane = 1;  
final int zPlane = 2;
```

```
Control C1 = new Control(xPlane);  
Control C2 = new Control(yPlane);  
Control C3 = new Control(zPlane);
```

Nenhuma thread criada ainda

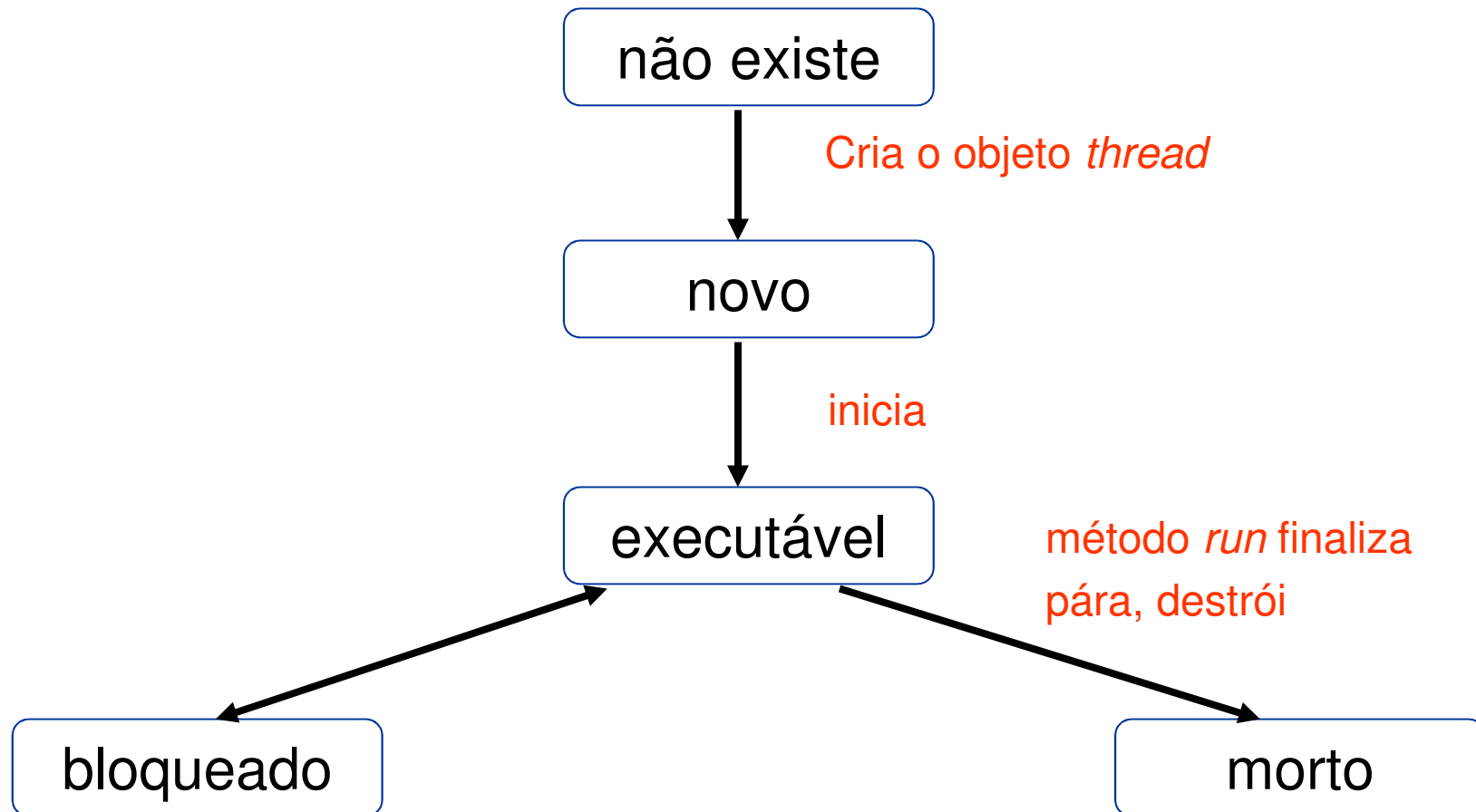
```
Thread X = new Thread(C1);  
Thread Y = new Thread(C2);  
Thread Z = new Thread(C2);
```

Construtores passaram uma interface *Runnable* e *threads* criadas

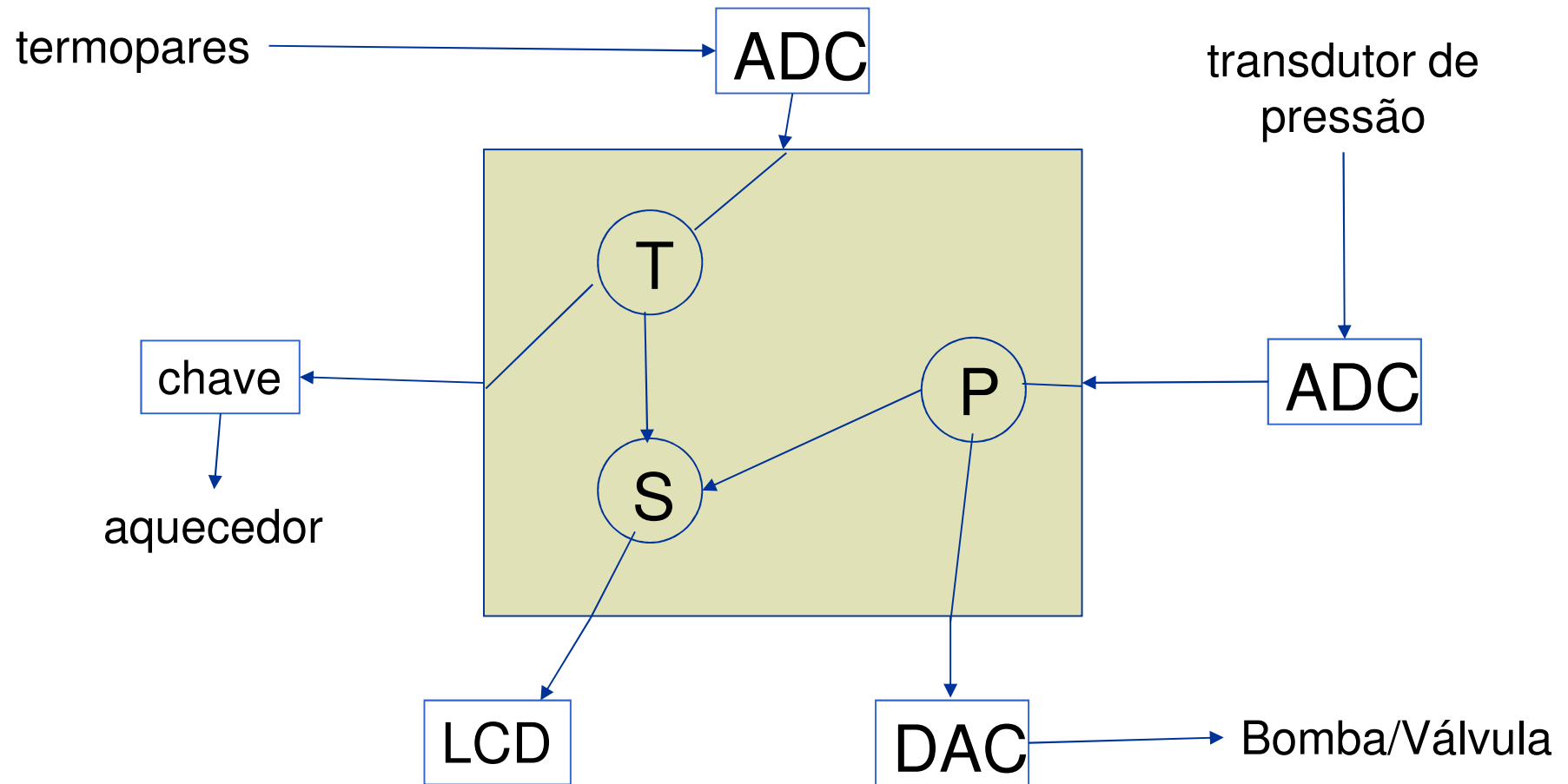
```
X.start();  
Y.start();  
Z.start();
```

Inicia *thread*

# Estados da *Thread* no Java



# Um Simples Sistema Embarcado



- Objetivo principal é manter a temperatura e pressão de algum processo químico dentro de limites definidos

# Possíveis Arquiteturas de Software

- Um único programa que ignora a concorrência lógica de T, P e S; nenhum suporte do SO é necessário
- T, P e S são escritos em uma linguagem de programação sequencial (ou como programas separados ou procedimentos distintos no mesmo programa) e as primitivas do sistema operacional são usadas para criação e interação do programa/processos
- Um único programa concorrente contém a estrutura lógica de T, P e S; nenhum suporte do SO é necessário embora exija-se um suporte de tempo de execução

Qual é a melhor abordagem?

# Pacotes Úteis

```
package Data_Types is  
    type Temp_Reading is new Integer range 10..500;  
    type Pressure_Reading is new Integer range 0..750;  
    type Heater_Setting is (On, Off);  
    type Pressure_Setting is new Integer range 0..9;  
end Data_Types;
```

Definições  
de tipo  
necessário

```
with Data_Types; use Data_Types;
```

```
package IO is  
    procedure Read(TR : out Temp_Reading); -- from ADC  
    procedure Read(PR : out Pressure_Reading);  
    procedure Write(HS : Heater_Setting); -- to switch  
    procedure Write(PS : Pressure_Setting); -- to DAC  
    procedure Write(TR : Temp_Reading); -- to screen  
    procedure Write(PR : Pressure_Reading); -- to screen  
end IO;
```

Procedimento  
para troca de  
dados com o  
ambiente

# Procedimentos de Controle

```
with Data_Types; use Data_Types;
package Control_Procedures is
    -- procedures for converting a reading into
    -- an appropriate setting for output.
    procedure Temp_Convert(TR : Temp_Reading;
                           HS : out Heater_Setting);
    procedure Pressure_Convert(PR : Pressure_Reading;
                                PS : out Pressure_Setting);
end Control_Procedures;
```



# Solução Sequencial

```
with Data_Types; use Data_Types; with IO; use IO;
with Control_Procedures; use Control_Procedures;

procedure Controller is
  TR : Temp_Reading;   PR : Pressure_Reading;
  HS : Heater_Setting; PS : Pressure_Setting;
begin
  loop
    Read(TR);          -- from ADC
    Temp_Convert(TR,HS);
    Write(HS);         -- to switch
    Write(TR);         -- to screen
    Read(PR);
    Pressure_Convert(PR,PS);
    Write(PS);
    Write(PR);
  end loop; -- infinite loop
end Controller;
```

Nenhum SO é necessário

# Desvantagens da Solução Sequencial

- As leituras de temperatura e pressão devem ser feitas na mesma taxa
  - O uso de contadores e *ifs* irão melhorar a solução
- Ainda pode ser necessário dividir os procedimentos de conversão `Temp_Convert` e `Pressure_Convert`, e intercalar suas ações de forma a atender a um equilíbrio necessário de carga
- Enquanto atendendo a uma leitura da temperatura, nenhuma atenção é dada a pressão (e vice versa)
- Um falha do sistema que resulta (por exemplo) no controle nunca retornar da leitura de temperatura significa que nenhuma chamada para leitura de pressão será realizada

# Um Sistema Melhorado

```
procedure Controller is
  TR : Temp_Reading;   PR : Pressure_Reading;
  HS : Heater_Setting; PS : Pressure_Setting;
  Ready_Temp, Ready_Pres : Boolean;
begin
  loop
    if Ready_Temp then
      Read(TR); Temp_Convert(TR, HS);
      Write(HS); Write(TR);
    end if;
    if Ready_Pres then
      Read(PR); Pressure_Convert(PR, PS);
      Write(PS); Write(PR);
    end if;
  end loop;
end Controller;
```

O que está errado  
com esta solução?

# Problemas

- A solução é mais confiável
- Mas o programa passa uma elevada proporção do seu tempo em um **laço ocupante** checando os dispositivos de entrada (*polling*)
- Espera-ocupantes são inaceitavelmente ineficiente
- Programas que dependem de espera ocupantes são difíceis de projetar, compreender e provar corretude

O programa sequencial não leva em consideração que os ciclos de pressão e temperatura são subsistemas independentes. Em um programa concorrente, ambos subsistemas são representados por tarefas

# Usando as Primitivas do SO (1)

```
package OSI is
  type Thread_ID is private;
  type Thread is access procedure;

  function Create_Thread(Code : Thread)
    return Thread_ID;
  -- other subprograms
  procedure Start(ID : Thread_ID);
private
  type Thread_ID is ...;
end OSI;
```

# Usando a Primitivas do SO (2)

```
package Processes is
    procedure Temp_C;
    procedure Pressure_C;
end Processes;

with ...;
package body Processes is
    procedure Temp_C is
        TR : Temp_Reading;  HS : Heater_Setting;
    begin
        loop
            Read(TR); Temp_Convert(TR, HS);
            Write(HS); Write(TR);
        end loop;
    end Temp_C;
```

# Usando a Primitivas do SO (3)

```
procedure Pressure_C is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
begin
    loop
        Read(PR) ;
        Pressure_Convert (PR, PS) ;
        Write(PS) ;
        Write(PR) ;
    end loop;
end Pressure_C;
end Processes;
```

# Usando a Primitivas do SO (4)

```
with OSI, Processes; use OSI, Processes;  
procedure Controller is  
    TC, PC : Thread_ID;  
begin  
    TC := Create_Thread(Temp_C'Access);  
    PC := Create_Thread(Pressure_C'Access);  
    Start(TC);  
    Start(PC);  
end Controller;
```

Melhor, solução  
mais confiável

Para SO realístico,  
solução torna-se  
ilegível!



# Abordagem usando Tarefas no Ada

```
with ...;
procedure Controller is
  task Temp_Controller;
  task body Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
  begin
    loop
      Read(TR);
      Temp_Convert(TR, HS);
      Write(HS); Write(TR);
    end loop;
  end Temp_Controller;
begin
  null;
end Controller;
```

```
task Pressure_Controller;
task body Pressure_Controller is
  PR : Pressure_Reading;
  PS : Pressure_Setting;
begin
  loop
    Read(PR);
    Pressure_Convert(PR, PS);
    Write(PS); Write(PR);
  end loop;
end Pressure_Controller;
```

# Vantagens de uma Abordagem Concorrente



- Tarefas do controlador executam concorrentemente: cada um contém um loop indefinido no qual o ciclo de controle é definido
- Enquanto uma tarefa está suspensa (esperando por uma leitura) a outra pode executar
  - se forem ambas suspensas um laço ocupante não é executado
- A lógica da aplicação é refletida no código
  - o paralelismo inerente do domínio é representado por executar concorrentemente as tarefas do programa
- Ambas as tarefas enviam dados para a tela
  - Exclusão mútua para acesso a tela (terceira entidade é necessário)

# Resumo



- **seção crítica** — código que deve ser executado sob exclusão mútua
- **sistema produtor-consumidor** — dois ou mais processos trocando dados via um buffer finito
- **espera ocupante** — um processo continuamente checando uma condição para ver se é possível proceder
- **livelock** — uma condição de erro no qual um ou mais processos são proibidos de progredir enquanto consome ciclos de processamento
- **deadlock** — uma coleção de processos suspensos que não podem proceder

# Resumo



- **semáforo** — um inteiro não negativo que só pode ser modificado por procedimentos atômicos WAIT e SIGNAL
- **POSIX mutexes** e **variáveis de condição** fornecem monitores com interface procedural