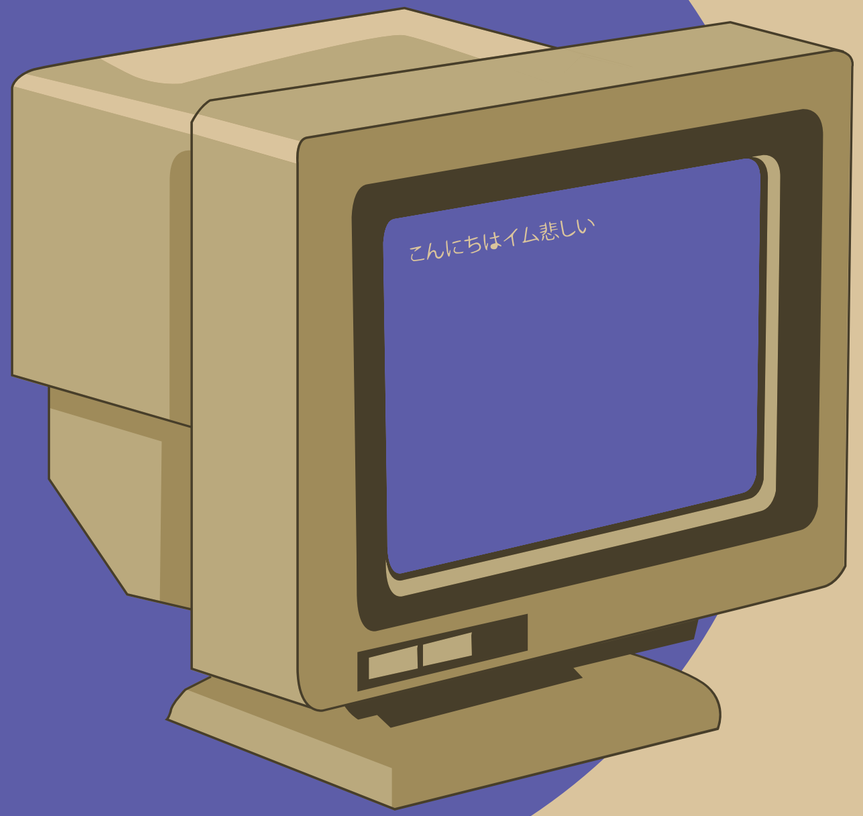
GRUPO 05

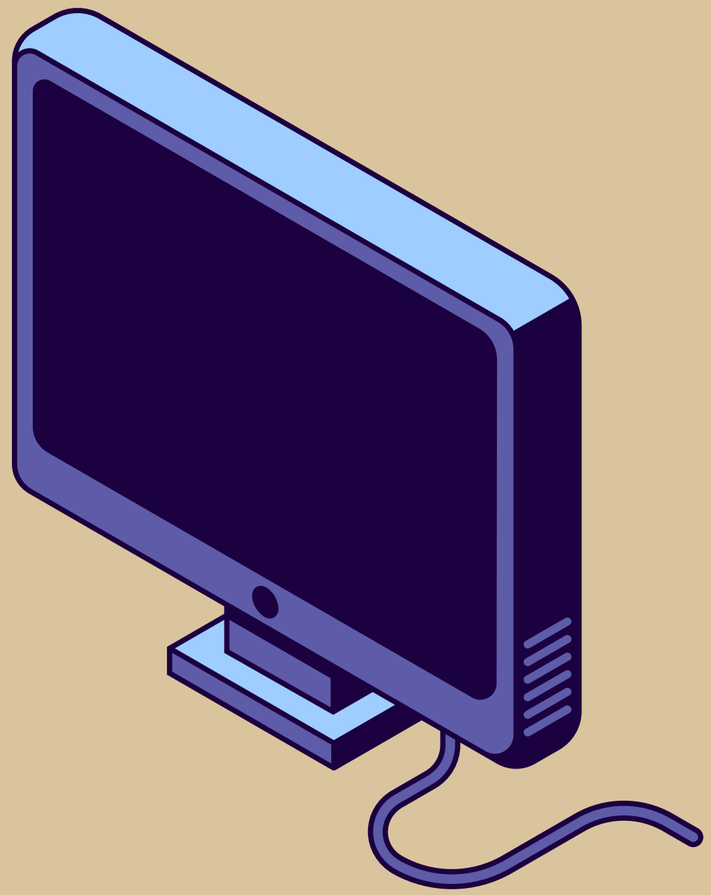



# IDENTIFICAÇÃO DE DEADLOCK USANDO ALGORITMO DO BANQUEIRO

WWW.DCC-UFRR.APP

GRUPO 05

# DEADLOCK

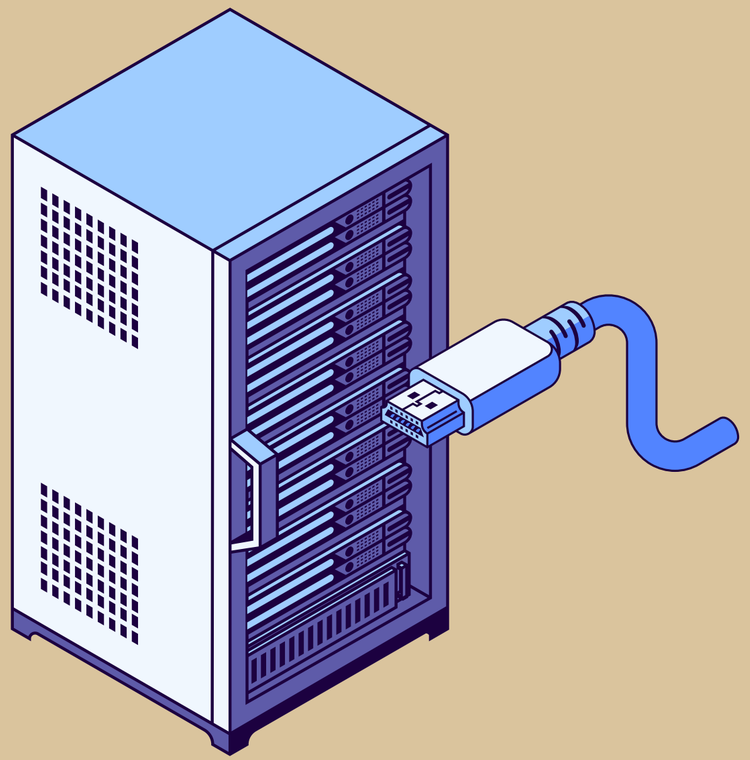


Em sistemas operacionais e em ambientes de computação concorrente. Refere-se a uma situação em que dois ou mais processos estão impedidos de prosseguir porque cada um está esperando por um recurso que está sendo retido por outro processo.

Isso ocorre principalmente quando dois processos se encontram, sendo que existe uma espera permanente devido a necessidade da utilização de recursos de outro. Assim sendo, se torna um looping de solicitações e esperas.

GRUPO 05

# ALGORITMO DO BANQUEIRO



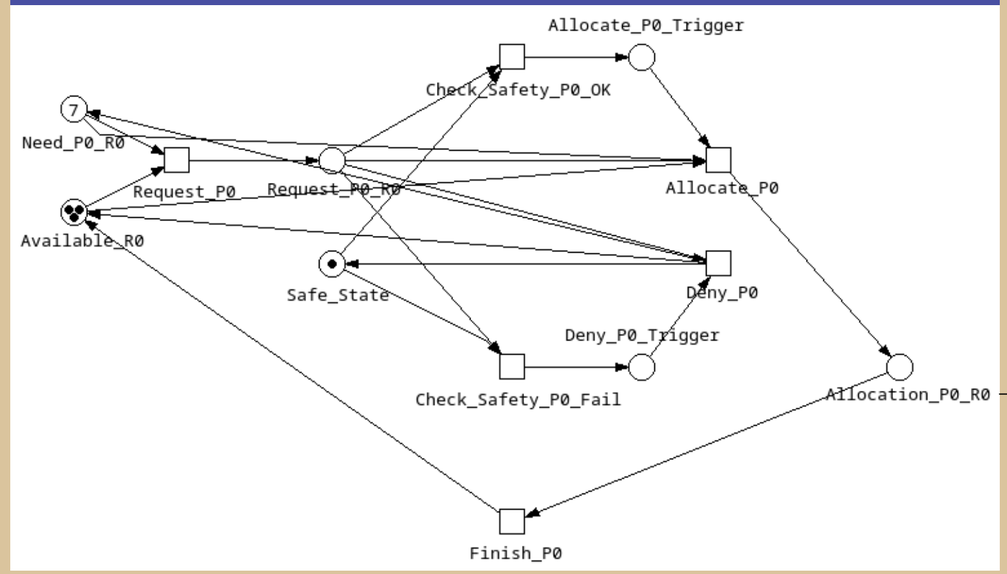
**1.** **Verificação dos pedidos dos processos**
O processo solicita recursos ao sistema, o mesmo verifica se não excede a máxima declarada pelo processo e se os recursos estão disponíveis.

**2.** **Simulação e alocação**
Após verificação, o sistema simula alocação, verificando se essa alocação é segura para execução do processo, assim pensando em finalizar e não deadlock

**3.** **Decisão de conceder ou não o pedido**
Após simulação, se o estado for seguro, os recursos são efetivamente alocados ao processo. Caso não, o processo deve esperar!

GRUPO 5

# REDE DE PETRI

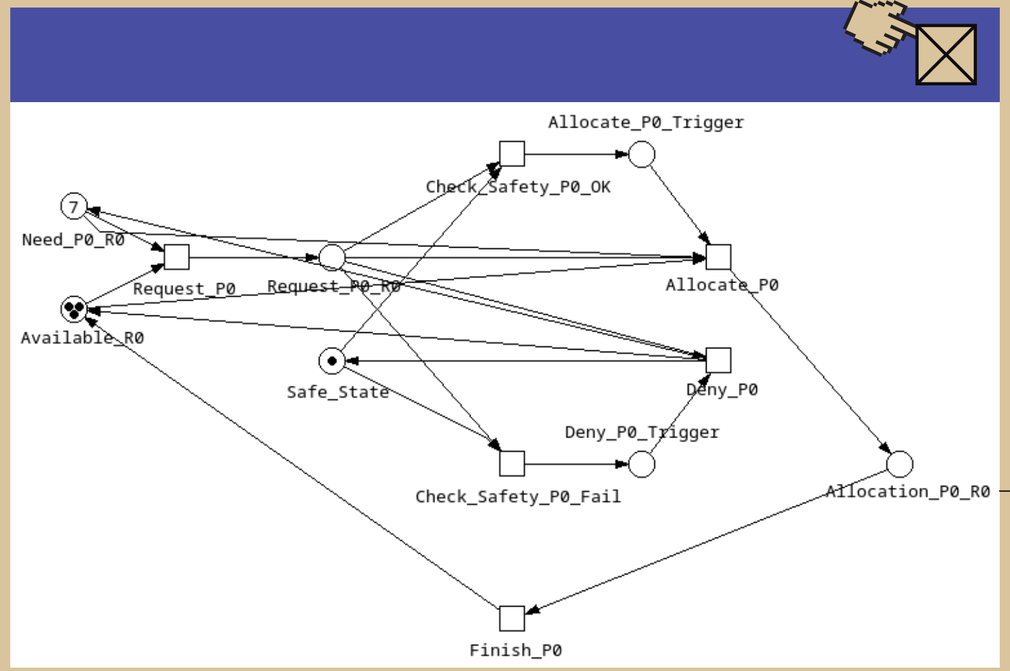



A rede de Petri modela o comportamento do Algoritmo do Banqueiro com foco em um único processo (P0) e um recurso (R0). Os lugares representam o número de recursos disponíveis, as requisições, as alocações, as necessidades e o estado de segurança do sistema. As transições simulam as ações do processo: fazer requisição, verificar a segurança, alocar o recurso, negar a requisição ou finalizar o uso.

A transição Check_Safety_P0_OK simula a lógica do algoritmo que só permite alocação se o sistema permanecer em um estado seguro, evitando deadlocks. Caso contrário, a transição Check_Safety_P0_Fail ativa a negação da requisição.

GRUPO 5

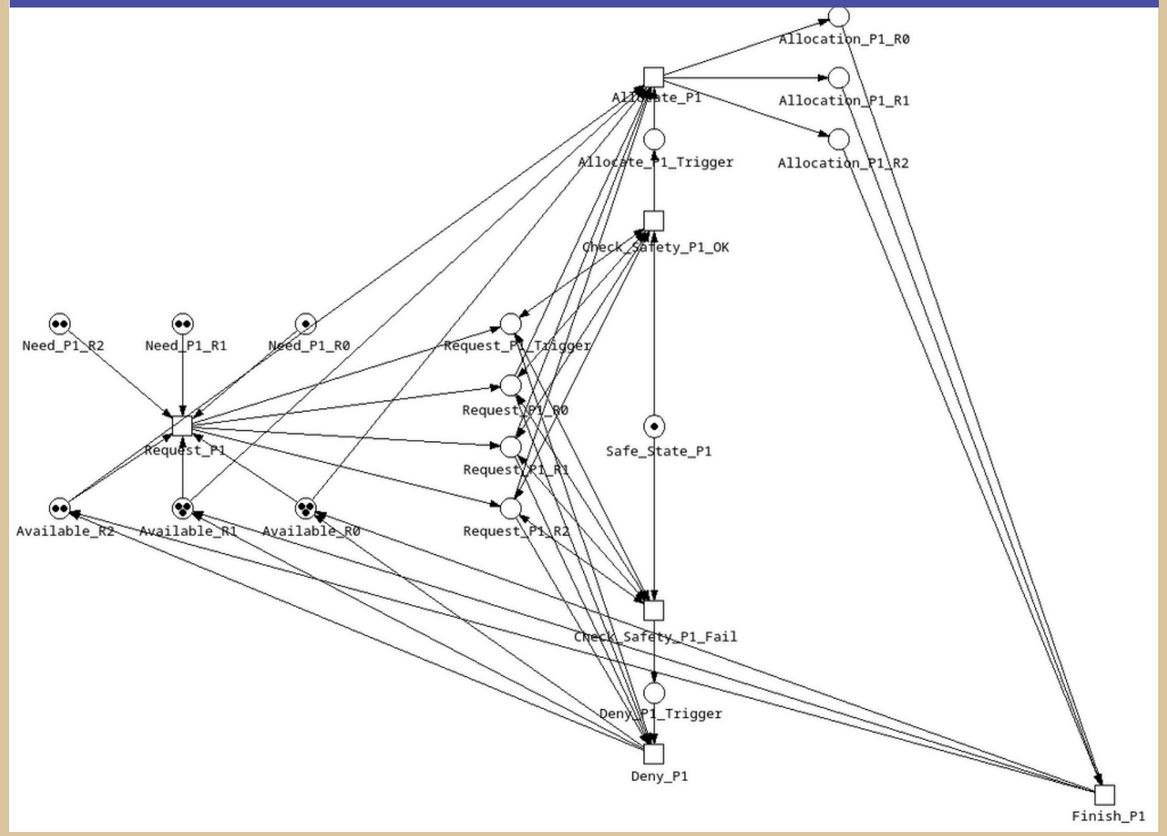
# ANÁLISE COM TINA




- A rede é limitada (bounded), ou seja, não há crescimento indefinido de marcas — comportamento esperado para controle de recursos.
- O grafo de alcançabilidade possui 24 classes (estados alcançáveis).
- O sistema não é vivo, indicando que, em certas situações, o processo pode ficar sem ações disponíveis (por exemplo, aguardando recurso sem possibilidade de avanço).
- Há possível reversibilidade, sugerindo que certos estados podem levar de volta ao estado inicial, dependendo da sequência de transições.
- 5 classes mortas foram identificadas, o que reforça a importância da verificação de segurança feita pelo algoritmo.

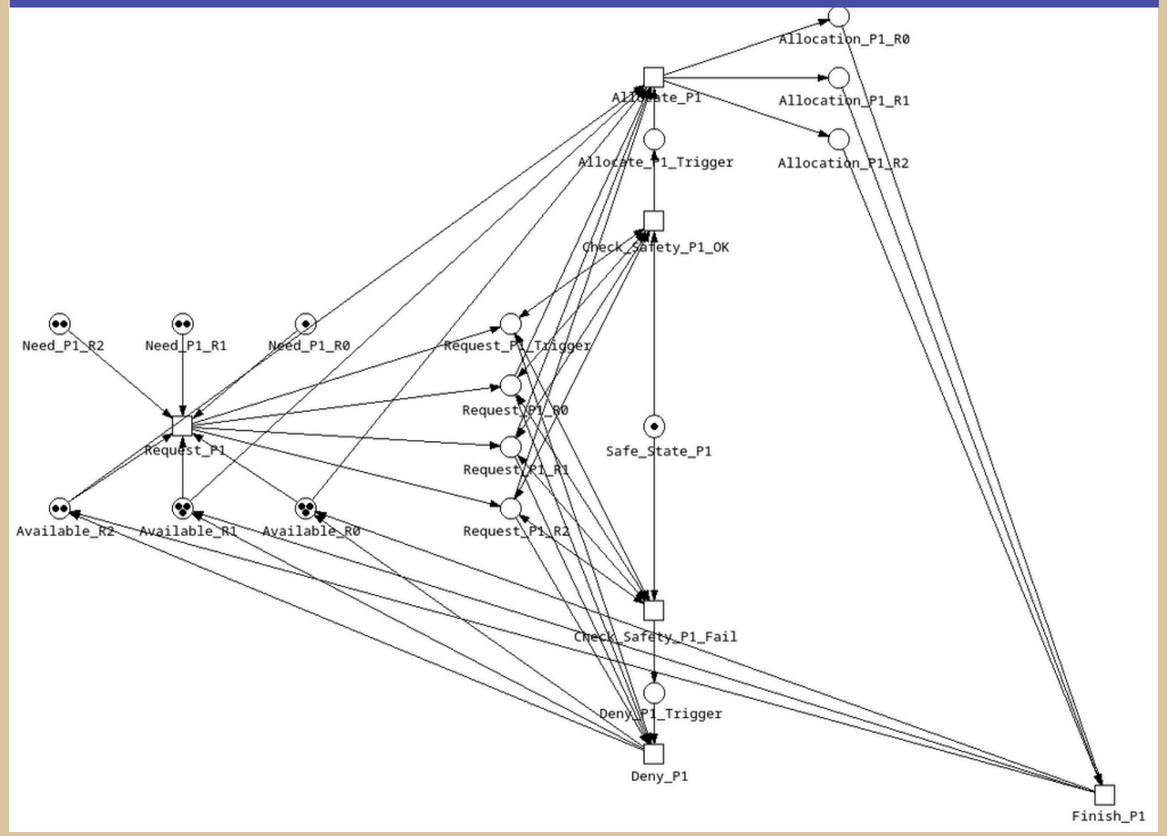
GRUPO 5

# REDE DE PETRI CONCORRÊNCIA




Esta rede modela o comportamento do algoritmo do banqueiro considerando múltiplos recursos (R0, R1, R2) sendo utilizados por um processo concorrente (P1). A estrutura da rede permite representar os estados de requisição, verificação de segurança, alocação e liberação dos recursos simultaneamente. A transição Request_P1 representa a solicitação conjunta de todos os recursos necessários, e as transições Check_Safety_P1_OK e Check_Safety_P1_Fail simulam a verificação do estado seguro do sistema antes de efetuar a alocação.

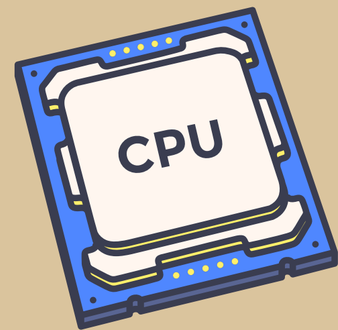
GRUPO 5




# ANÁLISE COM TINA

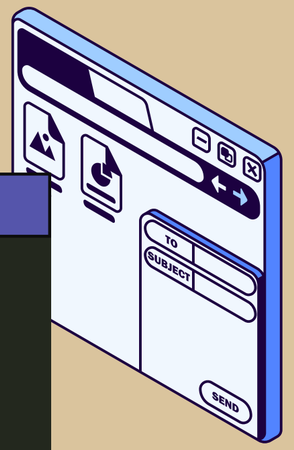
A análise da rede no TINA indicou que o sistema é limitado (bounded), ou seja, os lugares da rede não acumulam infinitos tokens, o que garante controle sobre os recursos. No entanto, a rede não é viva (not live), o que significa que, dependendo da sequência de transições, pode-se alcançar estados onde nenhuma transição está habilitada — sugerindo possíveis bloqueios. Ainda assim, o modelo é possivelmente reversível, indicando que há caminhos que permitem retornar ao estado inicial.

O baixo número de classes alcançadas (apenas 7) evidencia que o espaço de estados do sistema é reduzido, o que pode ser consequência de limitações na solicitação ou alocação concorrente. A modelagem serve como base para identificar potenciais condições de deadlock ao lidar com múltiplos recursos em ambientes concorrentes.



GRUPO 05

# IMPLEMENTAÇÃO




```c
#include <stdio.h>
#include <stdbool.h>

#define P 5 // Número de processos
#define R 3 // Número de recursos

// Dados do sistema
int available[R] = {3, 3, 2}; // Recursos disponíveis
int max[P][R] = {             // Máximo de cada processo
    {7, 5, 3},
    {3, 2, 2},
    {9, 0, 2},
    {2, 2, 2},
    {4, 3, 3}
};

int allocation[P][R] = {      // Recursos já alocados
    {0, 1, 0},
    {2, 0, 0},
    {3, 0, 2},
    {2, 1, 1},
    {0, 0, 2}
};
int need[P][R];               // Recursos necessários (max - allocation)

// Função para calcular a matriz need
void calculateNeed() {
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }
}
```

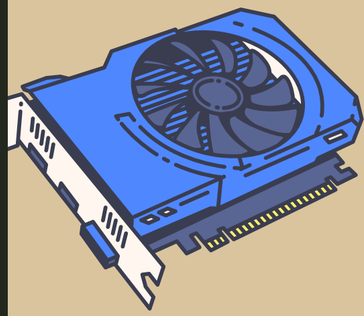
```c
// Verifica se o sistema está em estado seguro
bool isSafe() {
    int work[R];
    bool finish[P] = {0};

    // Inicializa work com os recursos disponíveis
    for (int i = 0; i < R; i++) {
        work[i] = available[i];
    }

    int safeSeq[P];
    int count = 0;

    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            if (!finish[p]) {
                int j;
                for (j = 0; j < R; j++) {
                    if (need[p][j] > work[j])
                        break;
                }

                if (j == R) {
                    // Processo pode ser finalizado
                    for (int k = 0; k < R; k++)
                        work[k] += allocation[p][k];

                    safeSeq[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }

        if (!found) {
            printf("O sistema nao esta em estado seguro!\n");
            return false;
        }
    }

    printf("O sistema esta em estado SEGURO!!\nSequencia segura: ");
    for (int i = 0; i < P; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n\n");
    return true;
}
```

GRUPO 05

# IMPLEMENTAÇÃO

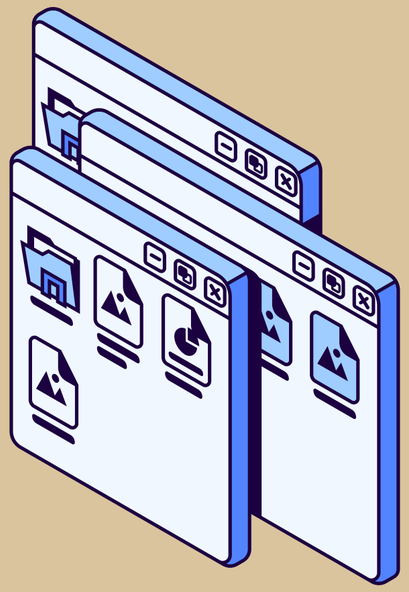
```c
int main() {
    calculateNeed();
    isSafe();

    // Exemplo de solicitação
    int request1[R] = {1, 5, 2}; // P1 solicita recursos
    requestResources(1, request1);

    int request2[R] = {3, 4, 0}; // P4 solicita recursos
    requestResources(2, request2);

    return 0;
}
```

```c
// Função para solicitar recursos
bool requestResources(int processID, int request[]) {
    printf("Processo P%d solicita recursos: ", processID);
    for (int i = 0; i < R; i++)
        printf("%d ", request[i]);
    printf("\n\n");

    // Verifica se pedido não excede o máximo necessário
    for (int i = 0; i < R; i++) {
        if (request[i] > need[processID][i]) {
            printf("Erro: Processo pediu mais do que o necessario.\n");
            return false;
        }
    }

    // Verifica se recursos estão disponíveis
    for (int i = 0; i < R; i++) {
        if (request[i] > available[i]) {
            printf("Erro: Recursos insuficientes.\n");
            return false;
        }
    }

    // Faz alocação temporária (simulação)
    for (int i = 0; i < R; i++) {
        available[i] -= request[i];
        allocation[processID][i] += request[i];
        need[processID][i] -= request[i];
    }

    // Verifica se sistema continua seguro
    if (isSafe()) {
        printf("Recursos foram alocados.\n\n");
        return true;
    }

    // Caso não esteja seguro, desfaz a alocação
    for (int i = 0; i < R; i++) {
        available[i] += request[i];
        allocation[processID][i] -= request[i];
        need[processID][i] += request[i];
    }

    printf("Recursos nao foram alocados para evitar deadlock.\n");
    return false;
}
```



GRUPO 5

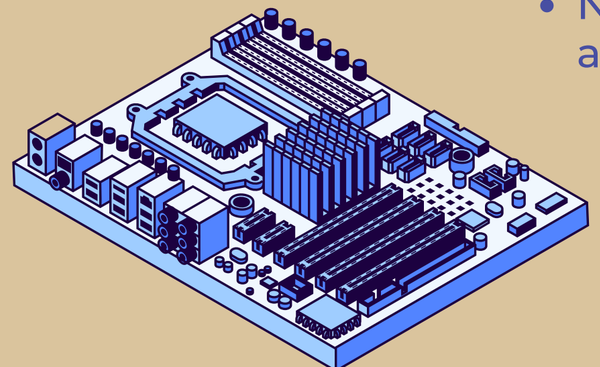
# TESTES COM O ALGORITMO DO BANQUEIRO EM C



Para verificar a correta detecção de estados seguros e possíveis deadlocks, realizamos testes simulando requisições de processos a partir de uma configuração inicial. Os testes foram feitos com 5 processos e 3 tipos de recursos. A matriz need foi calculada como max - allocation.

## CONFIGURAÇÃO INICIAL:

- Available: {3, 3, 2};
- Max: definida para cada processo com base na demanda máxima;
- Allocation: indica os recursos atualmente alocados;
- Need: obtida automaticamente a partir das duas anteriores.



GRUPO 5

# RESULTADOS DOS TESTES

## TESTE 1 – ESTADO INICIAL:

Sem realizar nenhuma requisição, executamos o algoritmo para verificar se o sistema já inicia em um estado seguro.
Resultado: o sistema está seguro, com a seguinte sequência viável de execução:
P1 → P3 → P4 → P0 → P2

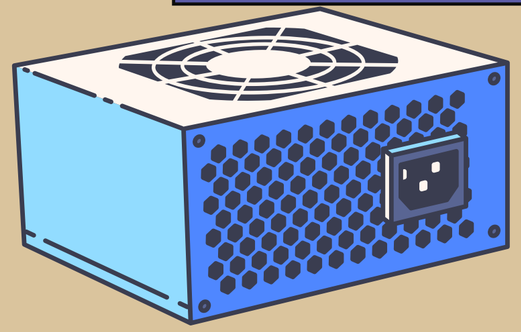
## TESTE 2 – REQUISIÇÃO DE P1:

P1 solicitou {1, 0, 2}. O pedido foi validado (menor ou igual a need e available), a alocação foi simulada e o algoritmo foi reexecutado.
Resultado: o sistema continuou seguro, mantendo uma sequência possível semelhante à anterior. Recursos foram alocados com sucesso.

## TESTE 3 – REQUISIÇÃO DE P4:

P4 solicitou {3, 3, 0}. O pedido excedia os recursos disponíveis no momento (Available[0] = 2).
Resultado: requisição negada. O algoritmo corretamente impediu a alocação, mantendo o sistema em segurança.

GRUPO 5



# OBRIGADO PELA A ATENÇÃO