Mecanismo (dispatcher): *como* **salvar e restaurar o contexto — as rotinas que fazem a operação.** **Dispatcher** (despachante) salva registradores de A no TCB(A).
O **scheduler** (quando consultado) decide qual tarefa deve rodar a seguir (política) **Política (scheduler)**: *quem* será o próximo a executar — regras (RR, SJF, prioridades, etc.).
TCB (do inglês Task Control Block) Novo → Pronto → Execução → Bloqueado → Terminado
Como o `fork()` funciona: O processo pai invoca `fork()`;
O SO cria uma **cópia quase idêntica** do processo pai; Ambos continuam executando, mas `fork()` retorna valores diferentes: Para o pai: retorna o **PID do filho**; Para o filho: retorna **0**. Depois de criado, o processo filho normalmente usa **`exec()`** para **substituir seu código** por outro programa (por exemplo, um shell executando um comando do usuário).
Término e coleta de processos: `wait()` e `exit()`.Quando um processo termina, ele chama **`exit()`**. O sistema operacional **mantém suas informações temporariamente** até que o pai chame **`wait()`** para "coletar" o status de término.
->Esse processo morto, que já terminou mas ainda não foi coletado, é chamado de zumbi.
Sem hierarquia: Melhor isolamento (Um processo não impacta muito no outro). A responsabilidade dogerenciamento do processo, cabe a aplicação resolver isso. Necessita de melhor gerenciamento.
Com hierarquia: Simplicidade de controle de processo. Porém, se o programa não fizer corretamente o wait() para recolher o estado dos filhos, podem surgir processos zumbis;
Processo Zumbi:Estado: "Terminado, aguardando coleta".
->Ocupa entrada na tabela de processos
->Não consome CPU nem memória de execução, mas pode acumular-se se o pai nunca chamar `wait()`.

Tanenbaum explica que o SO precisa manter o código de saída do processo morto, pois o pai pode querer saber se o filho terminou com sucesso ou falhou.

Modelo	N:1	1:1	N:M
Resumo	N *threads* do processo mapeados em uma *thread* de núcleo	Cada *thread* do processo mapeado em uma *thread* de núcleo	N *threads* do processo mapeados em M< N *threads* de núcleo
Implementação	no processo (biblioteca)	no núcleo	em ambos
Complexidade	baixa	média	alta
Custo de gerência	baixo	médio	alto
Escalabilidade	alta	baixa	alta
Paralelismo entre *threads* do mesmo processo	não	sim	sim
Troca de contexto entre *threads* do mesmo processo	rápida	lenta	rápida
Divisão de recursos entre tarefas	injusta	justa	variável, pois o mapeamento *thread*→ processador é dinâmico
Exemplos	GNU Portable Threads, Microsoft UMS	Windows, Linux	Solaris, FreeBSD KSE

Característica	Com processos	Com *threads* (1:1)	Híbrido
Custo de criação de tarefas	alto	baixo	médio
Troca de contexto	lenta	rápida	variável
Uso de memória	alto	baixo	médio
Compartilhamento de dados entre tarefas	canais de comunicação e áreas de memoria compartilhada.	variáveis globais e dinâmicas.	ambos.
Robustez	alta, um erro fica contido no processo.	baixa, um erro pode afetar todas as *threads*.	média, um erro pode afetar as *threads* no mesmo processo.
Segurança	alta, cada processo pode executar com usuários e permissões distintas.	baixa, todas as *threads* herdam as permissões do processo onde executam.	alta, *threads* que necessitam as mesmas permissões podem ser agrupadas em um mesmo processo.
Exemplos	Servidor Apache (versões 1.*), SGBD PostGres	Servidor Apache (versões 2.*), SGBD MySQL	Navegadores Chrome e Firefox, SGBD Oracle

Para que o sistema operacional possa suspender e retomar a execução de tarefas de forma transparente (sem que as tarefas o percebam), é necessário definir operações para salvar o contexto atual de uma tarefa em seu TCB e restaurá-lo mais tarde no processador. Por essa razão, o ato de suspender uma tarefa e reativar outra é denominado uma **troca de contexto**.
Processos órfãos: Ocorre quando um processo pai termina antes de seus filhos. O sistema, então, reatribui esses filhos a um processo especial (geralmente o init ou systemd no Linux).

Escalonadores: Objetivos e métricas (tempo de espera, resposta, retorno, utilização da CPU);
○ Primeiro a chegar, primeiro a servir - FCFS (First-Come, First-Served); **Lote**
○ Menor tarefa primeiro - SJF (Shortest Job First); **Lote**
○ Tempo restante menor primeiro - SRTF (Shortest Remaining Time First); **Lote**
○ Chaveamento Circular - RR (Round Robin); **Interativo**
○ Escalonamento por prioridades. **Tempo real**

->**Sinais**: simples e assíncronas.
->Um **pipe** é uma via unidirecional de comunicação entre dois processos relacionados (pai e filho, geralmente).
->**Filas de mensagens (Message Queues)**As filas permitem enviar e receber mensagens com identificação, prioridade e tamanho variável. Elas não exigem parentesco entre processos (diferente dos pipes). Cada mensagem tem um tipo, e o processo receptor pode filtrar o tipo que quer ler.
->**Memória compartilhada** (Shared Memory): É o mecanismo mais rápido, pois permite que múltiplos processos acessem a mesma região de memória física.
->**S*ockets*** são uma generalização dos pipes, capazes de atravessar redes.

O que é uma condição de corrida?: É uma situação em que o resultado final depende da **ordem de execução** entre múltiplas tarefas (processos/threads) que acessam dados ou recursos compartilhados. Se essa ordem muda, o comportamento/do resultado muda — e isso é indesejado quando queremos consistência.
Região crítica (critical section): É o trecho de código que acessa recursos compartilhados e que deve ser executado por **no máximo uma tarefa de cada vez**. A propriedade chave que queremos é **exclusão mútua**: enquanto uma tarefa está na região crítica, as outras devem ficar impedidas de entrar.

Um **semáforo** pode ser visto como uma variável composta s que contém uma fila de tarefas s.queue, inicialmente vazia, e um contador inteiro s.counter, cujo valor inicial depende de como o semáforo será usado. O conteúdo interno do semáforo não é diretamente acessível ao programador; para manipulá-lo devem ser usadas as seguintes operações atômicas:
down(s): decrementa o contador interno s.counter e o testa: se ele for negativo, a tarefa solicitante é adicionada à fila do semáforo (s.queue) e suspensa. Caso contrário, a chamada down(s) retorna e a tarefa pode continuar sua execução. Dijkstra denominou essa operação P(s) (do holandês probeer, que significa tentar).
up(s): incrementa o contador interno s.counter e o testa: um contador negativo ou nulo indica que há tarefa(s) suspensa(s) naquele semáforo. A primeira tarefa da fila s.queue é então devolvida à fila de tarefas prontas, para retomar sua execução assim que possível. Deve-se observar que esta chamada não é bloqueante: a tarefa solicitante não é suspensa ao executá-la. Essa operação foi inicialmente denominada V(s) (do holandês verhoog, que significa incrementar).
Mutex: Muitos ambientes de programação, bibliotecas de threads e até mesmo núcleos de sistema proveem uma versão simplificada de semáforos, na qual o contador só assume dois valores possíveis: livre (1) ou ocupado (0). Esses semáforos simplificados são chamados de **mutexes** (uma abreviação de mutual exclusion), semáforos binários ou simplesmente locks (travas).
Monitor: De certa forma, um **monitor** pode ser visto como um objeto que encapsula o recurso compartilhado, com procedimentos (métodos) para acessá-lo. No monitor, a execução dos procedimentos é feita com exclusão mútua entre eles. As operações de obtenção e liberação do mutex são inseridas automaticamente pelo compilador do programa em todos os pontos de entrada e saída do monitor (no início e final de cada procedimento), liberando o programador dessa tarefa e assim evitando erros.

Produtor/Consumidor: Produtor produz no buffer, comsumidor consome no buffer. Deve-se observar que o acesso ao buffer é bloqueante.

```
mutex mbuf ;        // controla o acesso ao buffer
semaphore item ;    // controla os itens no buffer (inicia em 0)
semaphore vaga ;    // controla as vagas no buffer (inicia em N)

task produtor ()
{
  while (1)
  {
    ...                 // produz um item
    down (vaga) ;       // espera uma vaga no buffer
    lock (mbuf) ;       // espera acesso exclusivo ao buffer
    ...                 // deposita o item no buffer
    unlock (mbuf) ;     // libera o acesso ao buffer
    up (item) ;         // indica a presença de um novo item no buffer
  }
}

task consumidor ()
{
  while (1)
  {
    down (item) ;       // espera um novo item no buffer
    lock (mbuf) ;       // espera acesso exclusivo ao buffer
    ...                 // retira o item do buffer
    unlock (mbuf) ;     // libera o acesso ao buffer
    up (vaga) ;         // indica a liberação de uma vaga no buffer
    ...                 // consome o item retirado do buffer
  }
}
```

Jantar dos filósofos:

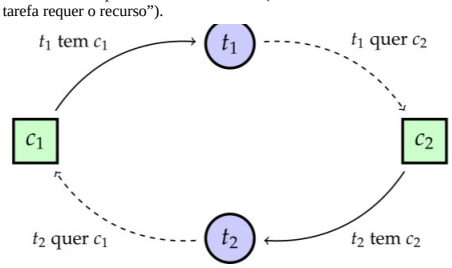
```
void take_forks(int i)          /* i: o número do filósofo, de 0 a N–1 */
{
    down(&mutex);               /* entra na região crítica */
    state[i] = HUNGRY;          /* registra que o filósofo está faminto */
    test(i);                    /* tenta pegar dois garfos */
    up(&mutex);                 /* sai da região crítica */
    down(&s[i]);                /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)               /* i: o número do filósofo, de 0 a N–1 */
{
    down(&mutex);               /* entra na região crítica */
    state[i] = THINKING;        /* o filósofo acabou de comer */
    test(LEFT);                 /* vê se o vizinho da esquerda pode comer agora */
    test(RIGHT);                /* vê se o vizinho da direita pode comer agora */
    up(&mutex);                 /* sai da região crítica */
}

void test(i)                    /* i: o número do filósofo, de 0 a N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Leitores e escritores
Problema: muitos leitores impedem os escritores

```
mutex marea ;               // controla o acesso à área
mutex mcont ;               // controla o acesso ao contador

int num_leitores = 0 ;      // número de leitores acessando a área

task leitor ()
{
  while (1)
  {
    lock (mcont) ;          // requer acesso exclusivo ao contador
    num_leitores++ ;        // incrementa contador de leitores
    if (num_leitores == 1) // sou o primeiro leitor a entrar?
      lock (marea) ;        // requer acesso à área
    unlock (mcont) ;        // libera o contador

    ...                     // lê dados da área compartilhada

    lock (mcont) ;          // requer acesso exclusivo ao contador
    num_leitores-- ;        // decrementa contador de leitores
    if (num_leitores == 0) // sou o último leitor a sair?
      unlock (marea) ;      // libera o acesso à área
    unlock (mcont) ;        // libera o contador
    ...
  }
}

task escritor ()
{
  while (1)
  {
    lock (marea) ;          // requer acesso exclusivo à área
    ...                     // escreve dados na área compartilhada
    unlock (marea) ;        // libera o acesso à área
    ...
  }
}
```

Deadlocks (Impasse): É uma situação onde duas ou mais threads ficam esperando indefinidamente uma pela outra para liberar um recurso (bloqueio ou lock) que a outra thread possui, resultando em um congelamento do programa.
Starvation (Inanição): Ocorre quando uma thread de baixa prioridade nunca consegue acesso a um recurso compartilhado porque threads de alta prioridade estão constantemente requisitando-o.
Condições para Impasse:
Exclusão mútua: o acesso aos recursos deve ser feito de forma mutuamente exclusiva,controlada por semáforos ou mecanismos equivalentes. No exemplo da contacorrente, apenas uma tarefa por vez pode acessar cada conta.
Posse e espera: uma tarefa pode solicitar o acesso a outros recursos sem ter de liberar os recursos que já detém. No exemplo da conta corrente, cada tarefa obtémo semáforo de uma conta e solicita o semáforo da outra conta para poderprosseguir.
Não-preempção: uma tarefa somente libera os recursos que detém quando assim o
decidir, e não os perde de forma imprevista (ou seja, o sistema operacional não retira à força os recursos alocados às tarefas). No exemplo da conta corrente, cada tarefa detém os mutexes obtidos até liberá-los explicitamente.
Espera circular: existe um ciclo de esperas pela liberação de recursos entre as tarefas envolvidas: a tarefa t1 aguarda um recurso retido pela tarefa t2 (formalmente, t1 → t2), que aguarda um recurso retido pela tarefa t3 , e assim por diante, sendoque a tarefa tn aguarda um recurso retido por t1 . Essa dependência circular pode ser expressa formalmente da seguinte forma: t1 → t2 → t3 → · · · → tn→ t1.

Em um **grafo de alocação de recursos** [Holt, 1972], as tarefas são representadas por círculos (○) e os recursos por retângulos (□). A posse de um recurso
por uma tarefa é representada como □ → ○ (lido como “o recurso está alocado à tarefa”), enquanto a requisição de um recurso por uma tarefa é indicada por○--> □ (lido como
“a tarefa requer o recurso”).



Prevenção: Exclusão mútua: se não houver exclusão mútua no acesso a recursos, não poderão ocorrer impasses. Ex: um servidor de impressão (spooling). A técnica de spooling previne impasses envolvendo as impressoras, mas não é facilmente aplicável a certos tipos de recurso, como arquivos em disco e áreas dememória compartilhada.
Posse e Espera: caso as tarefas usem apenas um recurso de cada vez, solicitando-o e liberando-o logo após o uso, impasses não poderão ocorrer. Ou requerer todos os processos antes de executar; Essa abordagem poderia levar as tarefas a reter os recursos por muito mais tempo que onecessário para suas operações, degradando o desempenho do sistema.
Não-preempção: Se um processo tomar a força um recurso de outro processo que tinha posse do mesmo, evitamos os deadlocks. No entanto, é de difícil aplicação sobre recursos como arquivos ou áreas de memória compartilhada, porque a preempção viola a exclusão mútua e pode provocar inconsistências no estado interno do recurso.
Espera Circular: Ao prevenir a formação de ciclos (de uma cadeia de dependência), impasses não poderão ocorrer. A estratégia mais simples para prevenir a formação de ciclos é ordenar todos os recursos do sistema de acordo com uma ordem global única, e forçar as tarefas a solicitar os recursos obedecendo a essa ordem.

Estratégias para resolver impasse:

Estratégia	Ideia central	Ponto forte	Limitação prática	Exemplo de uso real
Prevenção	Eliminar as condições de impasse	Garante segurança total	Reduz o desempenho e o uso eficiente dos recursos	Sistemas de controle crítico (aviões, usinas, equipamentos médicos)
Evitação	Permitir apenas alocações seguras (estado seguro)	Maior flexibilidade que a prevenção	Requer conhecer demandas futuras dos processos	Gerenciadores de transações, sistemas de banco de dados
Detecção e recuperação	Permitir impasses e corrigi-los depois	Usa recursos de forma eficiente	Recuperação pode causar perdas ou reinicializações	Sistemas gerais (Linux, Windows, UNIX)

Detecção e recuperação:
->Detecção com um único recurso por tipo: Se existir um ciclo no grafo de recursos, há um impasse.
->Detecção com múltiplos recursos por tipo: Existe um algoritmo parecido com o do banqueiro

Estratégia	Como funciona	Vantagens	Desvantagens
Eliminação	Encerra um ou mais processos do impasse	Simples, efetiva	Pode perder trabalho e dados
Liberação	Força processos a liberar recursos	Mantém alguns processos ativos	Pode causar inconsistências
Preempção	Retira temporariamente recursos	Evita matar processos	Difícil de implementar com segurança

Banqueiro:

Aspecto	Descrição
Ideia central	O sistema verifica se há recursos disponíveis. Se houver, simula conceder o pedido. Depois, testa se o sistema continuaria em um “estado seguro” — ou seja, se ainda existe uma ordem possível em que todos poderiam terminar. Se for seguro → concede o pedido Se não for → nega o pedido
Ponto forte	Evita impasses sem precisar matar processos.
Limitação prática	Exige saber previamente as necessidades máximas e consome muito tempo de cálculo.
Onde usar	Sistemas pequenos, previsíveis ou críticos (ex.: automação industrial, simulações educacionais).