Programação Multithread – Possuí memória compartilhada, comunicação por memória;
Programação Distribuida – Possuí memória distribuida, comunicação por troca de mensagens.

Imperativo: O programador declara quando criar uma nova tarefa (thread/processo);
Declarativo: O runtime ou compilador define quando criar uma nova thread.

Chaveamento de contexto – Salva-se os registradores e outras informações relevantes em um PCB, entra em estado ocioso (*ready* ou *blocked*) e chama outro processo em estado *ready*.

Threads – Processos lites (ID, PC e Stack reservados, CS, DS, recursos de OS compartilhados), podendo ser de usuário ou de kernel. O mapeamento de usuário kernel pode ser:
N-1: Gerenciamento feito pelo usuário. Não há processamento em multiplos cores. Um thread bloqueia todo um processo;
1-1: Cada user thread é mapeada para uma kernel thread. Permite execução paralela. Limita o número de threads que podem ser criadas (# de kernel threads). Criar thread de usuário executa chamada de kernel;
N-M: As threads são escalonadas durante a execução para qualquer kernel thread disponível. Permite execuçã em paralelo e inumeras user threads. Implementação mais complexa que as anteriores.

Memória compartilhada: operações via leitura-escrita;
Memória distribuida: troca de mensagens via send/recv.

Expressão de concorrência:
- Grafo de precedência; (grafo direcionado acíclico)
- Cobegin/Coend[1]; (t1; cobegin t2 | t3;t4 | coend; t5)
- funções S/P[1]; (sequencial: S(p,q); paralelo: P(p,q))
- Fork/Join; (a1: t1; …; fork a2; quit; […] ai: ti; …; join(an, ti); quit) […] an: quit;);
- Fok/Wait; (pid = fork(); wait(pid); exit(); //fork retorna pid do novo processo ou 0 caso seja o próprio processo criado)
- Notação de PC. (co t1 // t2 // .. // tn oc) ou (co [1.. n] { … })

1 Expressões limitadas a grafos propriamente aninhados.

Condicional: Uma tarefa só pode prosseguir após uma sinalização de condição;

Barreira: As tarefas devem sincronizar-se em um determinado momento.

Exclusão mútua: Composição de operações atômicas em um bloco de ações (sessão crítica).

Sessão crítica: Um bloco de ações que devem executar de forma sequencial em relação a tarefas que compartilham recursos; Caso não seja protegida, pode gerar problemas de corrida

Problema de corrida: múltiplas tarefas tentam acessar e manipular dados concorrentemente, alterando o valor dependendo da ordem acessada.

Propriedades esperadas em um programa concorrente:
- Safety propriety: Programa nunca entra em um estado ruim (Dois processos entrando ao mesmo tempo em uma sessão crítica) – garantido pela exclusão mútua;
- Liveness propriety: Programa sempre chega em um estado bom (aquele que todo o processo tem a chance de entrar na sessão crítica no momento apropriado) – garantido pela entrada na sessão crítica.

Atomicidade fina: Ação atômica a nível de hardware, “uma instrução”.
Atomicidade grossa: Ação atômica a nível de sotware, utilizando mecanismos de exclusão mútua.

Expressão parece atômica quando:
- É o único que acessa uma determinada variável (restrição forte) ou;
- Satisfaz *At-Most-Once*;

At-Most-Once (AMO) em **x=e**:
1 – Ou **e** possuí no máximo uma referência crítica e **x** não é lido por outro processo;
2 – Ou **e** não possuí nenhuma referência crítica e **x** pode ser lido por qualquer outro processo.

Primitiva <await(B) S;>: Espera B ser verdadeiro para executar S. Excluindo await(B) temos uma excusão mútua e excluindo o S temos uma sincronização condicional.
Caso B satisfaça AMO, await(B) == while(not B).

Propriedades desejadas a uma primitiva de exlusão mútua
1- Exclusão mútua (apenas 1 processo deve executar dentro da área de EM)
2- Progressão (um processo fora não pode impedir um processo de entrar na EM)
3- Espera Limitada (Todo processo deve esperar um tempo limitado para entrar na EM)
4- Não deve fazer suposição de velocidadade da execução ou do processador

Soluções em Software
- Algoritmo de Dekker v.5 e Petterson fazem a exclusão mútua para 2 processos usando variaveis de turno e de pretenção de entrada;
- Algoritmo de Lamport (Padaria) faz a exclusão mútua para N processos usando sistema de fichas, onde cada processo em paralelo recebe uma “senha” para entrar na sessão crítica, e eles vão sendo chamados em sequência.
Problemas de soluções em Software: Complexidade, *busy-wait*, desempenho, falta de clareza

Soluções em Hardware:
- Desativação de interrupções: Processos que estão na sessão crítica não podem ser preemptados

- Intruções tst-n-set e swap atômicas

busy-wait vs bloqueio: busy-wait faz computação desnecessária, ocupando o processador a toa. Alternativa de bloqueio traz o chaveamento de contexto como problema, mas sendo mais interessante para processadores monocore.

Inversão de prioridade: quando um processo menos prioritário impede um mais prioritário de prosseguir.

Mecanismos de sinconização:

Mutex:

Primitiva lock() libera um processo de entrar na sessão crítica caso possível, busy-wait caso contrário;

Primitiva unlock() libera sessão crítica.

Semáforo:

Importante: O semáforo conceitual não possuí regra de qual processo será desbloqueado.

Primitiva P(s) = Sse (s.cont > 0) s.cont -= 1 senão bloqueia e espera um recurso.

Primitiva V(s) = Se (existe proc. bloqueado) desbloqueia senão c += 1.

Variáveis de condição:

Uma variável é associada a uma fila de processos, NÃO tendo nenhum valor associado.

Primitiva Wait(cv): Bloqueia até que uma condição seja satisfeita.

Primitiva Signal(cv): Caso exista um processo na fila, desbloqueia-o senão é ignorado

Primitiva Signal-all(cv): Desbloqueia todos os processos na fila associada a essa variável

Monitores:

Pode ser visto como um objeto, onde os dados a serem protegidos são atributos privados, e os métodos são procedimentos que acessam esses dados são executados de forma mutuamente exclusiva.

Utiliza variaveis de condição para atrasar um procedimento até que uma condição seja verdadeira.

Possui uma fila de entrada para garantir um único processo acessando a SC, e uma fila de processos associada a cada CV.

Semântica do signal: Signal-n-Continue (não preemptiva e +usada), Signal-n-Wait, Signal-n-Urgent Wait (Preempta mas vai para o começo da fila de acesso).

Closed-call: ao chamar um segundo monitor, mantém exclusão mútua no primeiro. Menor concorrência, possibilidade de Deadlock.

Open-call: Libera o monitor e joga processo para a fila de espera. Maior complexidade, deve haver maior controle das variáveis do monitor.

Threads em Java:

1- Classe deve a) Extender a classe thread ou b) Implementar a interface Runnable
2- A Classe deve implementar o método run()
3- A instância s da Thread deve ser criada com new e chamar o método s.start();

Threads em C:

1- Criar uma variável de identificação
2- Usar p_thread_create(*identificador, atributos, função, parâmetros[]);

3- Usar p_thread_exit(*status) para terminar sua própria execução, p_thread_cancel(*ident) para acabar com a execução de uma outra thread e p_thread_join(*ident) para esperar uma thread.

Comunicação Inter-processos (IPC):
Utilizado quando não há a possibilidade/desejo de compartilhamento de memória;

Comunicação entre processos pode ser feito via memória compartilhada ou troca de mensagens
Mem compartilhada: Uma região de memória é estabelecida, e multiplos processos podem ler/escrever nele
Troca de mensagem: Um canal é estabelecido, e operações send/recv são feitas.

Pipes, sinais, fila de mensagens, shared-memory, Semaforos, Sockets, Websockets e Webservices
Pipes: Comunicação unidiriconal entre processos. Um processo lê outro, processo escreve em um arquivo. Deve ser compartilhado por processos com hierarquia

Sinais:
Um inteiro enviado por um processo que pode ser tratado por um processo receptor.
kill(n) é o comando para enviar um sinal para um processo.
signal(n, &f) é o comando para associar um handler a um processo.
O receptor pode:
1- Ignorar o Sinal (menos SIGSTOP e SIGKILL)
2- Tratar o sinal (associa um handler)
3- Toma uma ação padrão:
    a) abortar; b) memory dump; c) ignora; d) suspende o processo; e) continuar a executar.
Um processo só pode receber sinal de outro de um usuário de nível igual ou superior.
Tratamento feito por meio de uma mascara de bits.
isig() sempre é executado antes de bloquear e após acordar

Fila de mensagens:
Uma caixa postal de processos; um processo pode enviar uma mensagem com um tamanho variável a uma fila com id específico. Todos os receptores recebem a mensagem enviada e, caso não haja receptores, a mensagem entra na fila de enviadas ou o emissor na de emissores (caso n haja espaço para a mensagem)

Memória Compartilhada:
Um segmento de memória em que processos possam criar ponteiros e desanexar-se de uma região de memória.

TROCA DE MENSAGENS
Através de duas primitivas: send e recv; pode ser ponto-a-ponto ou multicast dependendo da semantica

Pontos desejados:
1- Simplicidade; 2- Uniformidade; 3- Eficiencia; 4- Confiabilidade; 5- Correção; 6- Flexibilidade; 7- Segurança; 8-Portabilidade
Mensagens de tamanho fixo dificultam uso e facilitam implementação, o contrário das de variável.

Nomeação direta (deve endereçar a um processo) ou indireta (processos compartilham caixa postal);
Comunicação síncrona (send e recv bloqueantes) ou assíncronas (send ou recv não bloquantes);

Buferização: caso não haja, send será sempre bloqueante. Caso haja, o send irá bloquear apenas se não houver espaço para bufferizar.

Comunicação síncrona (simples, mas concorrente): send e recv bloqueantes
Comunicação assíncrona (complexa, mas não concorrente): send não bloqueante, receive não importa, mas na prática é bloqueante.

Comunicação externa de dados:
Marshaling: transforma dados em um formato padrão;
Unsmarshaling: transforma dados em um formato local.
Usa um middleware ou biblioteca.

Falhas podem ser de omissão (quando alguém não faz sua parte), bizantina/arbitraria (alguma coisa fora do comum) ou de temporização (timeout)

Sockets:
Camada de transporte de redes. Omite as camadas inferiores de uma rede de computadores.
TCP: Orientado a conexão. (possui SYN,ACK e mais tudo que viu em redes)
UDP: Não orientado. (“só vai bruxão, eu n tenho nada a ver com isso”)
Portas: podem ser Conhceidas, Registradas ou Dinâmicas, Protocolos diferentes usam portas diferentes.
Portas são exclusivas a um processo;
Modelo Cliente (ativamente busca conexão e pede recurso) Servidor (espera passivamente conexão e fornece serviços).

Caracteristicas genéricas: Bidirecional, suporte a diversos protocolos.
A API Sockets pode:
criar (socket() - cria um descritor de socket)
nomear (bind() - Associa um descritor a um ip e porta específico – server only)
esperar conexão (listen() - Aguarda conexões – server TCP only).
aceitar conexão (accept() - Retorna um novo descritor de socket para aquela conexão específica – server TCP only)
conectar (connect() - Conecta-se a um socket específico – client (TCP) only)
envio (write()/sendto() - envia uma mensagem em um socket TCP/UDP)
recepção (read()/recvfrom() - recebe uma mensagem em socket TCP/UDP).
destruir (close() - fecha um socket)

UDP: send não bloqueante e receive bloqueante: o server faz um bind, e fica esperando para ler e depois enviar mensagens até ser fechado.
TCP: send e recv bloqueante apenas se não houver espaço no receptor ou menagens para o mesmo, respectivamente. Após o bind, o servidor deve fazer um listen e um accept para ficar no ciclo leitura escrita. Após o accept é que deve-se criar um novo processo ou thread.
Lembrar da network byte order!!

Confiabilidade: Para ter confiabilidade de comunicação, um protocolo request-reply deve implementar: abstração de alto nível, confirmações e timeout. Mensagens perdidas devem ser replicadas

Para a semantica do modelo Request-Reply:
At-most-once: não há ACKs: se uma mensagem é perdida, é perdida

At-least-once: Se um ack não é recebido, o remetente continua enviando pacotes e não são tratados pelo receptor como duplicados.
Exacly-once: Se um ACK não é recebido, o remetente continua tentando enviar. Se um pacote é duplicado, é tratado pelo receptor.

[Não-]Bloqueante: Referece a bloquear um processo até a recepção de um reply ou não.