# **Capítulo 1**

**Sistema Operativo**

Definição → Programa que atua como intermediário entre um usuário de um computador e o hardware do computador. Gerencia todos os recursos e controla e coordena o uso de hardware entre vários programas e usuários

Objetivos → Executar programas do usuário e facilitar a resolução de problemas do usuário, tornar o sistema do computador conveniente de usar, usar o hardware de maneira eficiente

Estrutura

→ Multiprogramação (necessária para eficiência)

* Execução intercalada de processos
* Várias tarefas são mantidas em memória simultaneamente e a CPU é partilhada entre elas
* Quando o programa atual tem de esperar, o processador pode executar outro programa

→ Multitasking (tempo partilhado)

* Extensão lógica na qual o CPU troca de tarefas com tanta frequência que os utilizadores podem interagir com cada tarefa enquanto está em execução, criando computação interativa
* Cada utilizador tem pelo menos um programa a executar na memória (processo)
* Se os processos não se enquadrarem na memória, swapping move-os para dentro e para fora para correr
* A memória virtual permite a execução de processos não completamente na memória

Operações

→ Interrupção de hardware por um dos dispositivos

→ Interrupção de software (exception ou trap)

→ Dual-mode operation - Permite que o SO se proteja a si próprio e a outros componentes do sistema (modo user e modo kernel (protege o acesso aos recursos do sistema))

O temporizador impede que uma aplicação tome conta do processador indefinidamente.

Gestão da atividade dos processos pelo SO

→ Criar e eliminar processos de utilizador e sistema

→ Suspender e retomar processos

→ Disponibilizar mecanismos para deadlock handling e para sincronização e comunicação de processos

Gestão do armazenamento pelo SO

→ Fornece uniformidade, visão lógica de armazenamento de informação

* Resumos propriedades físicas para unidade de armazenamento lógico - arquivo
* Cada meio é controlado por dispositivo (isto é, disco, unidade de fita)

**Sistema do Computador**

Componentes → Hardware, SO, programas e usuários

A maioria dos sistemas utiliza um único processador de uso geral.

Multiprocessadores

→ Vantagens

* Aumento da produção
* Economia de escala
* Maior fiabilidade – degradação graciosa ou tolerância a falhas

→ Dois tipos

* Multiprocessamento Assimétrico – cada processador é atribuído a uma tarefa específica.
* Multiprocessamento Simétrico – cada processador executa todas as tarefas

**Kernel**

Definição → Componente do SO responsável por gerir os recursos do sistema. Único programa que corre o tempo todo no computador.

**Interrupção**

Definição → Mecanismo que permite que o processamento normal de um processador seja interrompido. As interrupções são usadas para aumentar a eficiência, especialmente quando se usam componentes que operam a velocidades diferentes. Permitem que enquanto decorre uma operação de I/O de um processo, o processador continue a executar outros processos

Funções comuns → Uma interrupção transfere o controlo para a rotina de serviço de interrupção geralmente, através do interrupt vector, que contém os endereços de todas as rotinas de serviço

Uma trap ou exception é uma interrupção gerada pelo software causada por um erro ou uma solicitação do usuário. Um SO é interrupt driven.

**Armazenamento**

PByte TByte GByte MByte KByte Byte Bit

X8

X1024

X1024

X1024

X1024

X1024

Estrutura → Memória principal, armazenamento secundário, hard disks e solid-state disks (mais rápido que os hard disks)

Normalmente os discos são usados para armazenar dados que não cabem na memória principal ou que poderá ficar guardado por muito tempo.

**Estrutura de Acesso Direto à Memória**

→ Utilizado para dispositivos de I/O de alta velocidade capazes de transmitir informação a velocidades próximas da memória

→ Controlador de dispositivo transfere blocos de dados do buffer de armazenamento diretamente para a memória principal sem intervenção da CPU

→ Apenas uma interrupção é gerada por bloco, em vez da interrupção por byte

**Gestão**

→ Processo

* Programa de execução. Unidade de trabalho dentro do sistema. O programa é uma entidade passiva, o processo é uma entidade ativa.
* O processo necessita de recursos para cumprir a sua tarefa
* O término do processo requer a recuperação de quaisquer recursos reutilizáveis
* Processo Single-threaded tem um program counter que especifica a localização da próxima instrução a executar
* Processo Multi-threaded tem um program counter por thread
* Normalmente, o sistema tem muitos processos, alguns utilizadores, algum SO em execução simultânea em um ou mais CPUs

→ Memória

* Para executar um programa, todas (ou parte) das instruções/dados devem estar na memória
* A gestão da memória determina o que está na memória e quando
* Atividades de gestão da memória
  + - * Controlar que partes da memória estão atualmente a ser utilizadas e por quem;
      * Decidir que processos (ou partes destes) e dados entram e saem da memória
      * Alocar e desalocar espaço de memória, se necessário

→ Armazenamento

* Gestão do sistema de ficheiros
* Ficheiros são geralmente organizados em diretórios
* Acesso ao controlo da maioria dos sistemas para determinar quem pode aceder ao quê

**Proteção e Segurança**

→ Proteção

* Qualquer mecanismo de controlo do acesso de processos ou utilizadores a recursos definidos pelo sistema

→ Segurança

* Defesa do sistema contra ataques internos e externos

→ Sistemas geralmente distinguem primeiro entre os utilizadores, para determinar quem pode fazer o quê

**Ambientes Computacionais**

→ Tradicional

* Usa data centers físicos para armazenar recursos e executar um sistema de rede completo para operações diárias (permite acesso a dispositivos autorizados no servidor local)

→ Móvel

* Acesso a informações por meio de uma conexão de rede sem fio

→ Distribuído

* Conjunto de sistemas separados, possivelmente heterogéneos, numa rede

→ Client-Server

* Sistema responde a pedidos gerados pelos clientes

→ Peer-to-Peer

* Não distingue servidores de clientes. Cada um atua como cliente, servidor ou ambos

→ Virtualização

* Permite que os SO executem aplicativos em outros SO

→ Cloud Computing

* Fornece computação, armazenamento, até mesmo aplicativos como um serviço através de uma rede
* Extensão lógica da virtualização porque utiliza a virtualização como base para a sua funcionalidade

→ Sistemas Embutidos em Tempo Real

* Sistemas que monitorizam, respondem e controlam um ambiente externo conectado ao sistema do computador por meio de dispositivos de I/O. O sistema do computador deve atender às restrições temporais impostas pelo ambiente externo ao qual está conectado.

# **Capítulo 2**

**Sistema Operativo**

Serviços

→ SO proporcionam um ambiente de execução de programas e serviços a programas e utilizadores

→ Conjunto de serviços que fornecem funções úteis ao utilizador

* Interface do utilizador
* Execução do programa
* Operações I/O
* Manipulação do sistema de ficheiros
* Comunicações
* Deteção de erros

→ Conjunto de serviços que fornecem funções que garantem o funcionamento eficiente do sistema através da partilha de recursos

* Alocação de recursos
* Contabilidade
* Proteção e segurança

Interface do SO do utilizador

CLI

→ Permite a entrada direta no comando

* Às vezes implementado no kernel, às vezes por programas de sistemas
* Shells
* Às vezes vai buscar um comando do utilizador e executa-o
* Às vezes comanda built-in, às vezes apenas nomes de programas

GUI

→ Interface de ambiente de trabalho fácil de usar (rato, teclado, monitor, ícones, …)

Design e Implementação do SO

→ Não "solucionável", mas algumas abordagens têm-se revelado bem-sucedidas

→ Estrutura interna de diferentes SO podem variar muito

→ Iniciar o design definindo objetivos e especificações

→ Afetado pela escolha do hardware, tipo de sistema

→ Objetivos do utilizador

* SO deve ser conveniente de usar, fácil de aprender, fiável, seguro e rápido

→ Objetivos do sistema

* SO deve ser fácil de conceber, implementar e manter, bem como flexível, fiável, isento de erros e eficiente

→ A separação da política do mecanismo é um princípio muito importante, que permite a máxima flexibilidade para que as decisões políticas sejam alteradas mais tarde

* Política: O que será feito? (decidem o que será feito)
* Mecanismo: Como fazê-lo? (determinam como fazer algo)

Estrutura

→ Os SO de uso geral são um programa muito grande

→ Várias formas de estruturar

* Estrutura simples

MS-DOS

* Escrito para fornecer a maior funcionalidade no espaço mínimo
* Não dividido em módulos
* Embora tenha alguma estrutura, as suas interfaces e níveis de funcionalidade não estão bem separados
* Mais complexo

UNIX

* + - Limitado pela funcionalidade de hardware, o SO original UNIX tinha estruturação limitada
    - Composto por duas partes separáveis:
* Programas de sistemas
* Kernel (consiste em tudo abaixo da interface de chamada do sistema e acima do hardware físico; fornece o sistema de ficheiros, agendamento de CPU, gestão da memória e outras funções do sistema operativo; um grande número de funções para um nível)
* Em camadas – uma abstração
* SO dividido em várias camadas (níveis), cada uma construída em cima de camadas inferiores. A camada inferior (camada 0), é o hardware, o mais alto (camada N) é a interface do utilizador
* Com a modularidade, são selecionadas camadas de modo que cada um utilize funções (operações) e serviços de apenas camadas de nível inferior
* Microkernel – Mach
* Move o máximo do kernel para o espaço do utilizador
* A comunicação ocorre entre os módulos do utilizador utilizando a passagem de mensagens
* Benefícios
* Mais fácil alargar um microkernel
* Mais fácil de portar o SO para novas arquiteturas
* Mais fiável (menos código está a funcionar em modo kernel)
* Mais seguro
* Prejuízos
* Sobrecarga de desempenho do espaço do utilizador para o espaço de comunicação do kernel

Debugging

→ Encontrar e corrigir erros, ou bugs

→ Os SO geram ficheiros de registo que contenham informações de erro

→ Falha de uma aplicação pode gerar um ficheiro de core dump capturando a memória do processo

→ Falha no SO pode gerar um ficheiro de crash dump contendo memória kernel

→ Para além das falhas, o ajuste do desempenho pode otimizar o desempenho do sistema

Ajuste do desempenho

* Melhorar o desempenho removendo entraves
* O SO deve fornecer meios de computação e exibição de medidas de comportamento do sistema

**System Calls**

Definição

→ Interface de programação para os serviços prestados pelo SO

→ Normalmente escrita numa linguagem de alto nível (C ou C++)

→ Maioritariamente acessado por programas através de uma Interface de Programação de Aplicações de alto nível (API) em vez da utilização direta de chamadas do sistema

→ Exemplo: Copiar o conteúdo de um ficheiro para outro

Implementação

→ Normalmente, um número associado a cada chamada de sistema

→ A interface da chamada do sistema invoca a chamada do sistema pretendida no kernel do SO e devolve o estado da chamada do sistema e quaisquer valores de retorno

→ O caller não precisa de saber nada sobre como a chamada do sistema é implementada

* Só precisa de obedecer à API e entender o que o SO fará como resultado;
* A maioria dos detalhes da interface do SO é escondida do programador pela API

Passagem de parâmetros

→ Frequentemente, mais informações são necessárias do que simplesmente identidade da chamada do sistema desejada

→ Três métodos gerais utilizados para passar parâmetros ao SO

* Mais simples: passar os parâmetros nos registos
* Parâmetros armazenados num bloco, ou tabela, na memória, e endereço do bloco passado como parâmetro num registo
* Parâmetros colocados, ou empurrados, para a stack pelo programa e arrancados da stack pelo SO
* Métodos de bloco e stack não limitam o número ou comprimento dos parâmetros que estão a ser passados

Tipos

→ Controlo de processos

→ Gestão de ficheiros

→ Manutenção de informações

→ Comunicações

→ Proteção

**System Programs**

Definição

→ Proporcionam um ambiente conveniente para o desenvolvimento e execução de programas.

→ Podem ser divididos em

* Manipulação de ficheiros
* Informações de estado por vezes armazenadas numa modificação de ficheiros
* Suporte linguístico de programação
* Carregamento e execução de programas
* Comunicações
* Serviços de fundo
* Programas de aplicação

→ A visão da maioria dos utilizadores do SO é definida por programas do sistema, não pelas chamadas reais do sistema

**Módulos**

→ Muitos SO modernos implementam módulos de kernel carregados

* Usam uma abordagem orientada a objetos
* Cada componente do core é separado
* Cada um fala com os outros sobre interfaces conhecidas
* Cada um é carregado conforme necessário dentro do kernel

→ No geral, semelhantes às camadas, mas com mais flexibilidade (Linux, Solaris, etc)

**Hybrid Systems**

→ A maioria dos SO modernos não são, na verdade, um modelo puro

* Combina múltiplas abordagens para abordar as necessidades de desempenho, segurança, usabilidade
* Kernels Linux e Solaris no espaço de endereço do kernel, tão monolítico, além de modular para dynamic loading de funcionalidade
* Windows maioritariamente monolítico, mais microkernel para diferentes personalidades do subsistema
* Apple Mac OS X híbrido, em camadas, Aqua UI mais ambiente de programação Cocoa
* Abaixo está o kernel composto por partes de Mach microkernel e BSD Unix, além do kit de I/O e módulos dynamic loadable (chamadas extensões de kernel)

**System Boot**

→ O sistema operativo deve ser disponibilizado ao hardware para que o hardware possa iniciá-lo

→ O programa bootstrap (firmware) é carregado na inicialização ou reinicialização e normalmente é armazenado em ROM ou EPROM. Inicializa todos os aspetos do sistema. Carrega o kernel do SO, carrega-o na memória e inicia a execução

# **Capítulo 3**

**Processos**

Conceito

→ Um SO executa uma variedade de programas

* Batch System – tarefas
* Time-shared systems – programas ou tarefas de utilizadores

→ Programa de execução; execução de processo deve progredir de forma sequencial

→ Múltiplas partes

* O código do programa, também chamado de secção de texto
* Atividade atual, incluindo program counter, registos do processador
* Stack contendo dados temporários
* Secção de dados contendo variáveis globais
* Heap contendo memória dinamicamente alocada durante o tempo de execução

→ Programa é entidade passiva armazenada em disco (ficheiro executável), processo está ativo

→ Programa torna-se processo quando o ficheiro executável é carregado na memória (um programa pode ser vários processos)

Estado

→ Novo: O processo está a ser criado

→ Execução: As instruções estão a ser executadas

→ Esperando: O processo está à espera que algum evento ocorra

→ Pronto: O processo está à espera de ser atribuído a um processador

→ Terminado: O processo terminou a execução

Process Control Block (PCB)

→ Informação associada a cada processo

* Estado do processo
* Contador de programas – localização da instrução para a próxima execução
* Registos da CPU – conteúdos de todos os registos centrados no processo
* CPU agendando informações - prioridades, agendamento de ponteiros de fila
* Informação de gestão de memórias – memória alocada ao processo
* Informação contabilística – CPU usado, tempo do relógio decorrido desde o início, prazos limite
* Informações sobre o estado de I/O – dispositivos de I/O atribuídos ao processo, lista de ficheiros abertos

Process Scheduling

→ Maximiza a utilização do CPU, muda rapidamente os processos para a CPU para partilha de tempo

→ Seleciona entre os processos disponíveis para a próxima execução no CPU

→ Mantém filas de horários de processos

* Job queue – conjunto de todos os processos do sistema
* Ready queue – conjunto de todos os processos que residem na memória principal, prontos e à espera de executar
* Device queues – conjunto de processos à espera de um dispositivo de I/O
* Os processos migram entre as várias filas

Process Creation

→ Os processos pai criam processos filho que, por sua vez, criam outros processos, formando uma árvore de processos

→ Geralmente, o processo é identificado e gerido através de um process identifier (pid)

→ Opções de partilha de recursos

* Pais e filhos partilham todos os recursos
* Os filhos partilham um subconjunto de recursos dos pais
* Pais e filhos não partilham recursos

→ Opções de execução

* Pais e filhos executam simultaneamente
* Pais esperam até que os filhos terminem

→ Espaço de endereçamento

* Filho duplicado do pai
* O filho tem um programa carregado nele

→ Exemplos UNIX

* fork() cria novo processo
* exec() usada após um fork() para substituir o espaço de memória do processo por um novo programa

Process Termination

→ O processo executa a última declaração e pede ao SO que o elimine através da chamada do sistema exit()

* Devolve os dados do estado do filho para o pai (via wait())
* Os recursos do processo são desalocados pelo SO

→ O pai pode terminar a execução de processos filho usando a chamada do sistema abort(). Algumas razões para o fazer:

* O filho excedeu os recursos atribuídos
* A tarefa atribuída ao filho já não é necessária
* O pai está a sair e os SO não permitem que um filho continue sem o seu pai (cascading termination). O término é iniciado pelo SO

→ O processo pais pode aguardar o fim de um processo filho utilizando a chamada do sistema wait(). A informação do estado da chamada e o pid do processo encerrado

pid = wait(&status);

→ Se nenhum pai esperar (não invoca wait()) processo é um zombie

→ Se o pai terminou sem invocar wait, o processo é órfão

**Threads**

→ Até agora, o processo tem um único thread de execução

→ Considere ter vários program counters por processo

→ Vários locais podem ser executados de uma só vez

→ Deve então ter armazenamento para detalhes de thread, vários program counters no PCB

**Schedulers**

→ Scheduler de curto prazo (ou CPU scheduler) – seleciona qual o processo que deve ser executado em seguida e atribui CPU

→ Scheduler de longo prazo (ou job scheduler) – seleciona quais os processos que devem ser colocados na fila pronta; controla o grau de multiprogramação; esforça-se por uma boa mistura de processos

→ Os processos podem ser descritos como:

* Processo ligado a I/O – passa mais tempo a fazer I/O do que computações
* Processo ligado ao CPU – passa mais tempo a fazer cálculos

→ Scheduler de médio prazo – pode ser adicionado se o grau de programação múltipla precisar de diminuir

**Multitasking em Sistemas Móveis**

→ Alguns sistemas móveis permitem apenas um processo a funcionar, outros suspensos

→ Devido ao espaço do ecrã, os limites de interface do utilizador iOS oferecem um

* Processo de primeiro plano é controlado através da interface do utilizador
* Múltiplos processos de fundo
* Os limites incluem uma tarefa única e curta

→ Android corre em primeiro plano e plano de fundo, com menos limites

* O processo de fundo utiliza um serviço para executar tarefas
* O serviço pode continuar a funcionar mesmo que o processo de fundo seja suspenso
* O serviço não tem interface de utilizador, pequena utilização de memória

**Mudança de Contexto**

→ Quando o CPU mudar para outro processo, o sistema deve salvar o estado do processo antigo e carregar o estado guardado para o novo processo através de um context switch

→ Contexto de um processo representado no PCB

→ O tempo de mudança de contexto é superior; o sistema não faz nenhum trabalho útil durante a troca

* Quanto mais complexo o SO e o PCB mais longa a mudança de contexto

→ Tempo dependente do suporte de hardware

**Multiprocess Architecture – Chrome Browser**

→ Muitos navegadores da Web correram/correm como um único processo

→ Se um site causar problemas, todo o navegador pode travar ou falhar

→ O Google Chrome Browser é multiprocess com 3 tipos diferentes de processos:

* O processo do navegador gere a interface do utilizador, o disco e a rede I/O
* O processo de renderização renderiza as páginas web. Um novo renderizador é criado para cada website aberto
* Processo plug-in para cada tipo de plug-in

**Interprocess Communication (IPC)**

→ Os processos dentro de um sistema podem ser independentes ou cooperar

→ O processo de cooperação pode afetar ou ser afetado por outros processos, incluindo a partilha de dados

→ Razões para os processos de cooperação

* Partilha de informação
* Aceleração da computação
* Modularidade
* Conveniência

→ Os processos de cooperação necessitam de IPC

→ Dois modelos do IPC

* Memória partilhada
* Uma área de memória partilhada entre os processos que desejam comunicar
* Comunicação está sob o controlo dos processos dos utilizadores e não do SO
* Os principais problemas são fornecer mecanismos que permitam aos processos do utilizador sincronizar as suas ações quando acedem à memória partilhada.
* Passagem de mensagem
* Mecanismo para que os processos comuniquem e sincronizem as suas ações
* Sistema de mensagens – os processos comunicam uns com os outros sem recorrer a variáveis partilhadas
* O IPC fornece duas operações:
* send(menssage)
* receive(menssage)
* O tamanho da mensagem é fixo ou variável
* Se os processos P e Q quiserem comunicar, devem:
* Estabelecer uma ligação de comunicação entre eles
* Trocar mensagens através do send/receive
* Questões de implementação:
* Como são estabelecidas as ligações?
* Pode uma ligação ser associada a mais de dois processos?
* Quantas ligações podem existir entre cada par de processos de comunicação?
* Qual é a capacidade de uma ligação?
* O tamanho de uma mensagem que o link pode acomodar é fixo ou variável?
* É uma ligação unidirecional ou bidirecional?

**Indirect Communication**

→ As mensagens são direcionadas e recebidas a partir de caixas de correio (também designadas por ports)

* Cada caixa de correio tem um id único
* Os processos só podem comunicar se partilharem uma caixa de correio

→ Propriedades da ligação de comunicação

* Link estabelecido apenas se os processos partilharem uma caixa de correio comum
* Uma ligação pode estar associada a muitos processos
* Cada par de processos pode partilhar várias ligações de comunicação
* O link pode ser unidirecional ou bidirecional

**Problema Produtor-Consumidor**

→ Paradigma para processos de cooperação, processo de produção produz informação que é consumida por um processo de consumo

* Unbounded-buffer não coloca limite prático no tamanho do buffer
* Bounded-beffer assume que há um tamanho fixo do buffer

**Comunicações em Sistemas Produtor-Consumidor**

Sockets

→ Ponto final para a comunicação

→ Concatenação do endereço IP e do port – um número é incluído no início do pacote de mensagens para diferenciar os serviços de rede num anfitrião

→ O socket 161.25.19.8:1625 refere-se ao port 1625 no anfitrião 161.25.19.8

→ A comunicação consiste entre um par de sockets

→ Todos os ports abaixo de 1024 são conhecidos, usados para serviços padrão

→ Endereço IP especial 127.0.0.1 (loopback) para se referir ao sistema no qual o processo está a ser executado

Pipes

→ Atua como uma conduta que permite que dois processos se comuniquem

→ Questões

* A comunicação é unidirecional ou bidirecional?
* No caso de comunicação bidirecional, é metade ou full-duplex?
* Tem de existir uma relação (ou seja, pai-filho) entre os processos de comunicação?
* Os pipes podem ser utilizados numa rede?
* Ordinary Pipes
* Não podem ser acedidos fora do processo que o criou. Normalmente, um processo pai cria um pipe e usa-o para comunicar com um processo filho que criou
* Permitem comunicação estilo padrão Produtor-Consumidor
* O produtor escreve para uma extremidade (a extremidade de escrita do tubo)
* O consumidor lê do outro lado (write-end do pipe)
* São, portanto, unidirecionais
* Exige relação entre pai e filho entre processos de comunicação
* O Windows chama a estes pipes anónimos
* Named Pipes – podem ser acedidos sem uma relação pai-filho
* Os named pipes são mais poderosos que os ordinary pipes
* A comunicação é bidirecional
* Não é necessária qualquer relação pai-filho entre os processos de comunicação
* Vários processos podem usar o named pipe para comunicação
* Fornecido tanto nos sistemas UNIX como Windows

**Buffering**

→ Fila de mensagens anexadas ao link

→ Implementado de uma de três formas

* 1. Capacidade zero – Nenhuma mensagem está na fila de uma ligação

O remetente deve aguardar o recetor

* 2. Capacidade limitada – Comprimento finito de n mensagens

O remetente deve esperar se a ligação estiver completa

* 3. Capacidade ilimitada – Comprimento infinito

O remetente nunca espera

# **Capítulo 5**

**Sincronização de Processos**

Background

→ Os processos podem ser executados simultaneamente

* Podem ser interrompidos a qualquer momento, completando parcialmente a execução

→ O acesso simultâneo a dados partilhados pode resultar em inconsistência dos dados

→ A manutenção da consistência dos dados requer mecanismos para assegurar a execução ordenada dos processos de cooperação

→ Ilustração do problema:

* Suponha que quisemos fornecer uma solução para o problema do consumidor-produtor que preenche todos os buffers. Podemos fazê-lo tendo um contador inteiro que acompanha o número de buffers completos. Inicialmente, o contador é 0. É incrementado pelo produtor depois de produzir um novo buffer e é decrementado pelo consumidor depois de consumir um buffer.

Produtor Consumidor

while (true) { while (true) {  
 /\* produce an item in next produced \*/ while (counter == 0);

while (counter == BUFFER\_SIZE); /\* do nothing \*/

/\* do nothing \*/ next\_consumed = buffer[out];

buffer[in] = next\_produced; out = (out + 1) % BUFFER\_SIZE;

in = (in + 1) % BUFFER\_SIZE; counter--;

counter++; /\* consume the item in next consumed \*/

} }

Race Condition

→ counter++ → counter--

register1 = counter register2 = counter

register1 = register1 + 1 register2 = register2 - 1

counter = register1 counter = register2

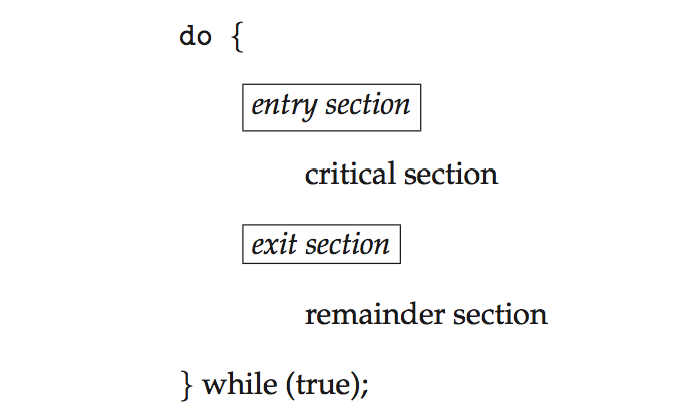
**Critical Section Problem**

→ Considere o sistema de n processos {p0, p1, ... pn-1}

→ Cada processo tem um segmento de código de critical section

→ Cada processo deve pedir permissão para introduzir critical section na entry section, pode seguir a critical section com a exit secction e, em seguida, a remainder section

Estrutura geral do processo Pi



Solução

→ Exclusão mútua

* Se o processo Pi estiver a executar na sua critical section, então nenhum outro processo pode ser executado nas suas critical secctions

→ Progresso

* Se nenhum processo estiver a ser executado na sua critical section e existirem alguns processos que desejem entrar na sua critical section, então a seleção dos processos que entrarão na critical section em seguida não pode ser adiada indefinidamente

→ Espera limitada

* Deve existir um limite no número de vezes que outros processos são autorizados a introduzir as suas critical sections após um processo ter feito um pedido para introduzir a sua critical section e antes que esse pedido seja concedido
* Assuma que cada processo executa a uma velocidade não zero
* Nenhuma suposição relativa à velocidade relativa dos processos n

No SO

→ Duas abordagens dependendo se o kernel é preventivo ou não

* Preventivo – Permite a preempção do processo ao correr em modo kernel
* Não preventivo – Corre até sair do modo kernel, bloqueia ou rende voluntariamente CPU

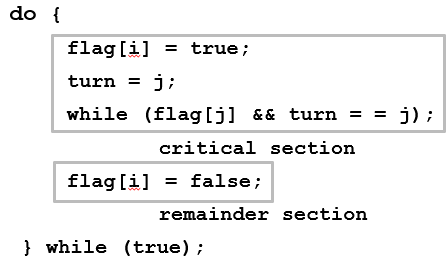
Nota: Definição preempção → Durante a execução simultânea de duas ou mais aplicações ou tarefas, aquilo que provoca mudança do processamento de uma aplicação ou tarefa para outra.

Solução Peterson

→ int turn - Indica de quem é a vez de entrar na critical secction

→ boolean flag[2] - Indica se um processo está pronto para entrar na critical secction

→ Os três requisitos da critical secction são preservados



**Sincronização de Hardware**

→ Muitos sistemas fornecem suporte de hardware para implementar o código de critical secction

→ Todas as soluções abaixo são baseadas na ideia de locking

* Proteger critical sections através de locks

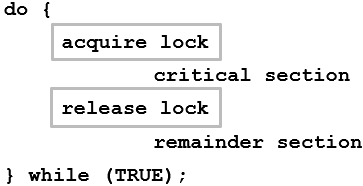
→ Uniprocessadores – podem desativar interrupções

* Atualmente em execução código seria executado sem preempção
* Geralmente demasiado ineficiente em sistemas multiprocessadores

→ Máquinas modernas fornecem instruções especiais de hardware atómico

* Atómico = não interrompível

Solução Problema CS Usando Locks



Instrução test\_and\_set

→ Definição

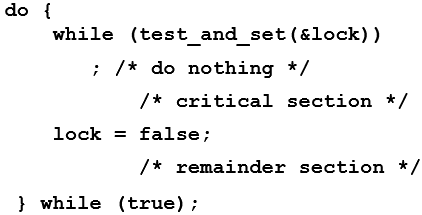
* Executado atomicamente
* Devolve o valor original do parâmetro passado
* Coloca o novo valor do parâmetro passado para "TRUE"

Text, letter

Description automatically generated

→ Solução usando test\_and\_set()

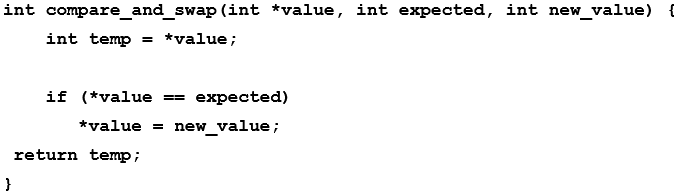
* Variável lock booleana partilhada, inicializada para “FALSE”



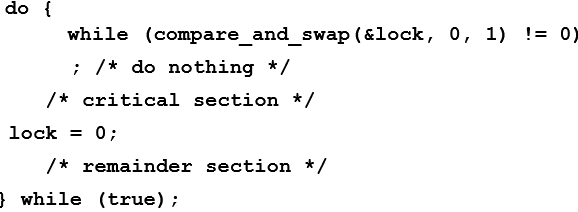
Instrução compare\_and\_swap

→ Definição

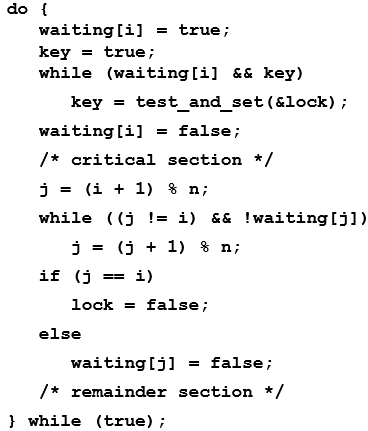
* Executado atomicamente
* Devolve o valor original do parâmetro passado "value"
* Retorna a variável "value" do parâmetro "new\_value" mas apenas se "value" =="expected". Ou seja, a troca só ocorre sob esta condição.
* "lock" inteiro partilhado inicializado a 0



→ Solução usando compare\_and\_swap ()



Exclusão Mútua de Espera Limitada com test\_and\_set



**Mutex Locks**

→ As soluções anteriores são complicadas e geralmente inacessíveis aos programadores de aplicações

→ Designers de SO constroem ferramentas de software para resolver o problema de critical secction. A mais simples é mutex lock

→ Proteger uma critical secction através de primeiro aquire() (adquirir) um lock e, em seguida, release() (libertar) o lock

* Variável booleana indicando se o lock está disponível ou não

→ As chamadas para aquire() e release() devem ser atómicas

→ Mas esta solução requer busy wait

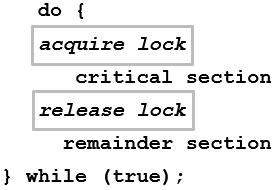
* Este lock é, portanto, chamado de spinlock

→ acquire() { → release() {  
 while (!available); available = true;

/\* busy wait \*/ }

available = false;

}



**Semáforos**

Definição

→ Ferramenta de sincronização que fornece formas mais sofisticadas (do que Mutex Locks) para o processo de sincronização das suas atividades

→ Semáforo S – variável inteira

→ Só pode ser acedido através de duas operações indivisíveis (atómicas)

* wait() e signal() – originalmente chamadas P() e V()

→ wait(S) { → signal(S) {  
 while (S <= 0); S++;

/\* busy wait \*/ }

S--;

}

Uso

→ Contagem de semáforos – o valor inteiro pode variar sobre um domínio sem restrições

→ Semáforo binário – o valor inteiro só pode variar entre 0 e 1

* O mesmo que uma Mutex Lock

→ Pode resolver vários problemas de sincronização

→ Considere P1 e P2 que exigem que S1 aconteça antes de S2

Criar um semáforo "synch" inicializado para 0

P1: P2:

S1; wait(synch);

signal(synch); S2;

→ Pode-se implementar um semáforo de contagem S como um semáforo binário

Implementação

→ Deve garantir que dois processos não possam executar wait() e signal() no mesmo semáforo ao mesmo tempo

→ Assim, a implementação torna-se no problema critical secction onde o código de wait e signal são colocados na critical secction

* Poderia agora ter busy waiting na implementação da critical secction
* Mas o código de implementação é curto
* Pouco busy waiting se a critical secction for raramente ocupada

→ Note que as aplicações podem passar muito tempo em critical secctions e, portanto, esta não é uma boa solução

Implementação sem Busy Waiting

→ A cada semáforo há uma fila de espera associada

→ Cada entrada numa fila de espera tem dois itens de dados

* valor (do tipo inteiro)
* apontador para o próximo registo na lista

→ Duas operações:

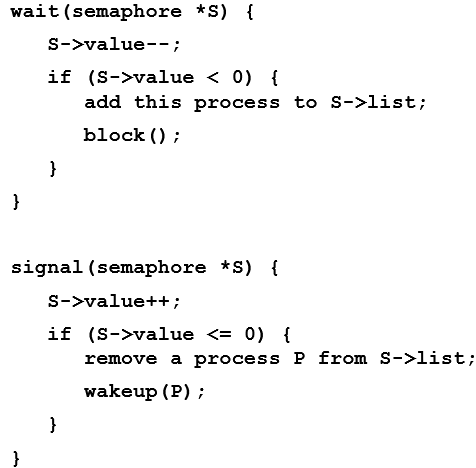
* block – colocar o processo invocando a operação na fila de espera apropriada
* wakeup – remover um dos processos na fila de espera e colocá-lo na fila pronta

→ typedef struct {

int value;

struct process \*list;

} semaphore;



**Deadlock and Starvation**

Deadlock

→ Dois ou mais processos aguardam indefinidamente um evento que pode ser causado por apenas um dos processos de espera

→ Seja S e Q dois semáforos inicializados a 1

P0 P1

wait(S); wait(Q);

wait(Q); wait(S);

... ...

signal(S); signal (Q);

signal (Q); signal(S);

Starvation (bloqueio indefinido)

→ Um processo nunca pode ser removido da fila do semáforo em que é suspenso

Inversão prioritária

→ Problema de agendamento quando o processo de menor prioridade detém um bloqueio necessário por um processo de maior prioridade

* Resolvido através de protocolo de herança prioritária

**Problemas Clássicos de Sincronização**

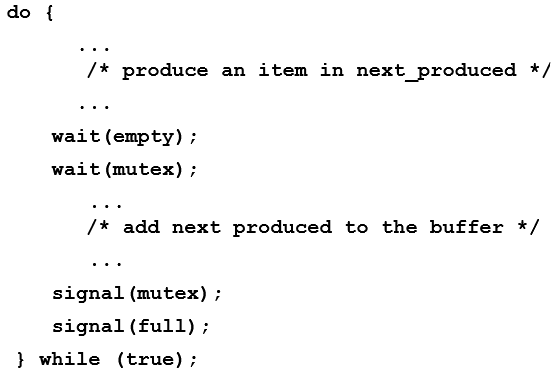
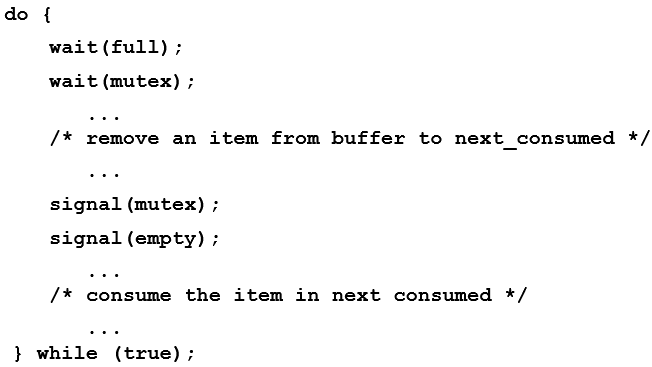
Bounded-Buffer Problem

→ n buffers, cada um pode segurar um item

→ Semáforo Mutex inicializado com valor 1

→ Semáforo full inicializado com valor 0

→ Semáforo empty inicializado com valor n

 Produtor Consumidor

Readers and Writers Problem

→ Um conjunto de dados é partilhado entre uma série de processos simultâneos

* Readers – Leem apenas o conjunto de dados; não realizam quaisquer atualizações
* Writers – Podem ler e escrever

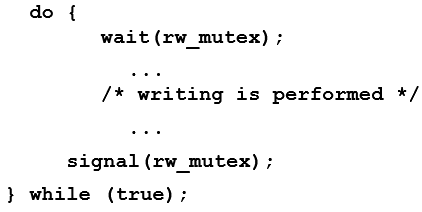
→ Problema – Permitir que vários readers leiam ao mesmo tempo

* Apenas um único writer pode aceder aos dados partilhados ao mesmo tempo

→ Várias variações de como os readers e writers são considerados – todos envolvem alguma forma de prioridades

→ Dados Partilhados

* Conjunto de dados
* Semáforo rw\_mutex inicializado a 1
* Semáforo mutex inicializado a 1
* Inteiro read\_count inicializado a 0

 Writer Reader

Text

Description automatically generated

→ Primeira variação do problema – nenhum reader fica à espera a menos que o writer tenha permissão para usar o objeto partilhado

→ Segunda variação do problema – uma vez que o writer está pronto, executa a escrita o mais rápido possível

→ Ambos podem ter starvation levando a ainda mais variações

→ O problema é resolvido em alguns sistemas através do kernel que fornece reader-writer locks

Dining-Philosophers Problem

→ Filósofos passam a vida alternando o pensamento e a comer

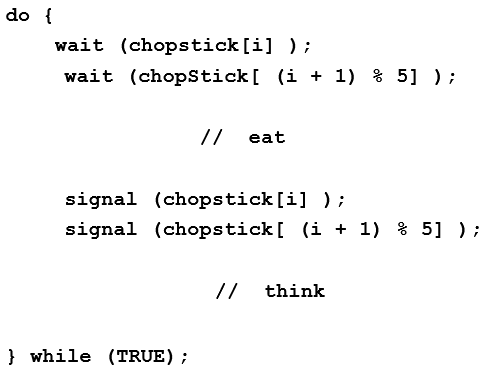
→ Não interagem com os vizinhos, ocasionalmente tentam pegar 2 pauzinhos (um de cada vez) para comer da tigela

* Precisa de ambos para comer, em seguida, liberta ambos quando feito

→ No caso de 5 filósofos

* Dados partilhados
* Tigela de arroz (conjunto de dados)
* Semáforo chopstick[5] inicializado a 1

→ Estrutura do Filósofoi



→ Tratamento do Deadlock

* Deixar um filósofo pegar nos pauzinhos apenas se ambos estiverem disponíveis (a colheita deve ser feita numa critical secction)
* Usar uma solução assimétrica. O filósofo numerado pega primeiro o pauzinho direito e depois no pauzinho esquerdo.

# **Monitores**

**Problemas com Semáforos**

→ Semáforo é uma primitiva de baixo nível e não é estruturada

→ É fácil cometer erros com semáforos:

* Esquecer de um release que emparelhe com um acquire
* Trocar ordem de acquires

→ Para sistemas grandes, a associação entre recursos e semáforos que os protegem pode não ser clara

→ Envolvem alguma “arte” na construção de soluções

→ Seria desejável uma primitiva que permita a construção sistemática de soluções

**Monitores**

→ Primitiva estruturada de controlo de concorrência

→ Tipo de dados com operações, que encapsula estado (semelhança com objetos)

→ Acesso concorrente é controlado internamente

→ Clientes podem simplesmente invocar operações

→ Apenas um processo pode estar “dentro” num dado momento

* Exclusão mútua é obtida trivialmente

→ Disponibiliza variáveis de condição

* Permitem processos bloquearem-se voluntariamente
* Usadas em problemas de ordem-de-execução (mas não só)

→ Entidade passiva usada por processos

→ Pode haver starvation na entrada

**Variáveis de condição** (condition queues)

Definição

→ Permitem a um processo bloquear-se voluntariamente

→ São declaradas explicitamente

→ O nome deverá sugerir uma condição (predicado) que se verdadeira permite ao processo prosseguir

→ Processos testam predicado sobre variáveis de estado do monitor e decidem se bloqueiam

→ Não têm valor que se leia ou escreva

Operações sobre variáveis de condição (monitores clássicos)

→ f – Fila de processos bloqueados associada a cada v.c.

→ p - Processo atual que executa num monitor mon

* Primitiva wait bloqueia processo na v.c.

wait(cond):

cond.f.append(p)

unlock(mon.mut)

suspend()

...

* wait liberta mutex antes de bloquear processo
* Primitiva signal liberta processo bloqueado na v.c

signal(cond):

if cond.f != []:

q = cond.f.pop(0)

ready(p)

* Se não existir processo bloqueado, o signal “perde-se” (ao contrário dos semáforos)

**Produtor-Consumidor com Bounded-Buffer como Monitor**

→ O buffer trata da exclusão mútua e ordem-de-execução

→ O código do produtor e consumidor fica trivial

Produtor Consumidor

while (…) { while (…) {

x = produce(); x = buffer.take();

buffer.put(x); consume (x);

} }

**Monitores Clássicos**

→ É garantido que

* Se estiver algum processo bloqueado num wait, a seguir a um signal, prossegue o processo bloqueado. Assim, se predicado é verdadeiro quando é feito signal o predicado permanece verdadeiro depois do wait.
* Mais tarde prossegue quem fez signal.
* Finalmente podem entrar no monitor outros processos

→ Este comportamento é chamado de imediate resumption requirement ou signal and urgent wait

**Signal All**

→ Disponibilização de variante de signal, signalAll

* Acorda todos os processos bloqueados na variável de condição
* Encontra-se em monitores modernos

**Monitores Modernos**

→ Os mais usados atualmente, por exemplo, em Java e Pthreads têm

E = W < S

* Ou seja
* Primeiro continua o processo que faz signal
* Depois pode correr o processo acordado ou pode correr um terceiro processo que estivesse a querer entrar;

→ Como um terceiro processo pode ter mudado o estado do monitor, o predicado pode já não ser verdadeiro depois do wait

→ Conclusão, temos que usar testes de predicados com while:

while (!predicado())

wait(cond);

**Monitores Modernos e Spurious Wakeups**

→ Às vezes, poderíamos ser tentados a não usar while

* Se não mudássemos o estado depois do signal e soubéssemos que mais nenhum processo pudesse estar a tentar entrar no monitor, não havendo perigo de ultrapassagem

→ Spurious Wakeups obrigam o uso de ciclos de espera

→ Para obter implementações eficientes de monitores em multiprocessadores, um wait pode, embora muito raramente, desbloquear mesmo sem ter sido feito signal

→ Conclusão, temos que fazer sempre espera em ciclo