Primeira Questão

SO como um facilitador de uso: Tem como objetivo tornar fácil e acessível para o usuário o proveito dos recursos de hardware. Para isto, deve ser interativo e útil na perspectiva do usuário, muitas vezes com uma interfaces de acesso para arquivos e programas, e prioriza a facilidade de uso em detrimento do controle da utilização dos recursos.

SO como um coordenador de recursos: Tem como foco o controle de uso dos recursos e de execução de programas para proporcionar segurança e eficiência às aplicações sem que elas conflitem entre si e causem erros. Consegue lidar com múltiplas solicitações de uso e conflitos. É seguro, robusto, escalonável e extensível.

SO como um Programa (kernel): É um programa semelhante aos outros - instruções sendo executadas na CPU - com a diferença de que é executado o tempo inteiro, e traz algum tipo de utilidade por si só. Todas as outras aplicações user-level devem executar em paralelo com ele e respeitando suas condições impostas.

Segunda Questão

O PCB do Linux possui: Estado do processo, número do processo (PID), PC (Program Counter), registradores da CPU, limite de memória e lista de arquivos abertos.

Quanto ao Mac OS X, tive dificuldade de achar informações claras, parece ter um certo nível de sigiloso, mas o que consegui identificar foi: ID de processo, credenciais, informação de sinal, outras informações UNIX (PC, registradores, estado do processo, etc), e por fim, associações entre uma User-thread e Machine-thread, abstrações feitas pelo próprio Mac OS X.

Ao meu ver, a do Linux parece ser mais objetiva e concisa quanto ao que é realmente essencial para implementar processos eficientes em um SO de sistema operacional de propósito-geral.

Terceira Questão

Threads a nível de Usuário são de fácil implementação, baratas de criar e trocar de contexto, mas em contraponto não são enxergadas pelo SO, e o bloqueio de uma causa o bloqueio de todo o processo. Dado isto, são independentes do SO e ideais para concorrência eficiente entre threads.

Threads a nível de Kernel tem uma implementação mais complicada e criação/troca de contexto mais cara, porém o SO tem ciência de sua presença, sendo assim capaz de gerenciar todas as threads de um processo e evitar bloqueios. Assim, seu uso é ideal para aplicações que frequentemente causam bloqueio, e necessitam de paralelismo na execução ou otimização dada pelo SO.

A thread pool pode ser utilizada tanto no espaço de Usuário quanto de Kernel, depende da biblioteca de threads e do Sistema Operacional, se ele suporta threads ou não. Sistemas que utilizam: Windows XP (Kernel), Java e C# (Bibliotecas a nível de Usuário).

Quarta Questão

O Mac OS X utiliza uma combinação dos algoritmos RR e Multilevel Feedback Queue. Assim, todas as tarefas têm a oportunidade de compartilhar CPU Time, e a maior prioridade sempre vai para as tarefas mais curtas e processos I/O-bound, maximizando a interação entre o usuário. No MFQ, os processos são divididos em categorias baseadas na sua necessidade da CPU, podendo ser promovidas por estar esperando muito tempo ou rebaixadas por excesso de utilização: Isto gera um equilíbrio de partilha de recurso, embora as Tasks I/O-bound sempre tenham prioridade nesta lógica. Como cada categoria é uma Queue, possui uma complexidade temporal de busca de tarefas inferior comparado à árvore balanceada do CFS do Linux.

O *CFS*, utilizado no Linux, procura balancear a utilização da CPU com a interação com o usuário (I-O) de forma justa para todas as tarefas. Assim como o Mac OS X, cumpre isso com a combinação de prioridades e divisão de tempos de execução meritocráticos para cada tarefa de forma fluída: Quanto menor o runtime de uma tarefa, maior será sua necessidade do processador, porém, ao invés de sempre dar alta prioridade para tarefas *I-O bound* como o Mac OS X faz, o CFS

promete tempo de execução justo para estas apenas quando precisarem (*sleeper fairness*). As tasks são armazenadas numa árvore balanceada O(log n), tornando este algoritmo mais performático que o do Mac OS X na escolha de tasks para execução.

Quinta Questão

O ideal é que muitos leitores possam ler ao mesmo tempo e não se bloqueiem entre si, mas que bloqueiem o acesso de escritores tentando entrar. Mas até mesmo nesta solução pode haver muitos leitores para poucos escritores, e acabar causando “starvation” para algumas threads.

A solução ideal é: Um escritor espera autorização de acesso à área crítica por um semáforo partilhado entre leitores e escritores, e, após ser autorizado e passar pela área crítica, sinaliza este mesmo semáforo para avisar as próximas threads que já saiu da seção crítica. Já um leitor espera o acesso à área crítica, e se recebe, incrementa e verifica se há outros leitores em *race-condition*: Se for o primeiro a entrar ali, manda um sinal para bloquear acesso dos escritores. Após sair, decrementa e verifica se ainda há leitores na área crítica, e se houver, sinaliza os escritores de que a seção crítica está livre.