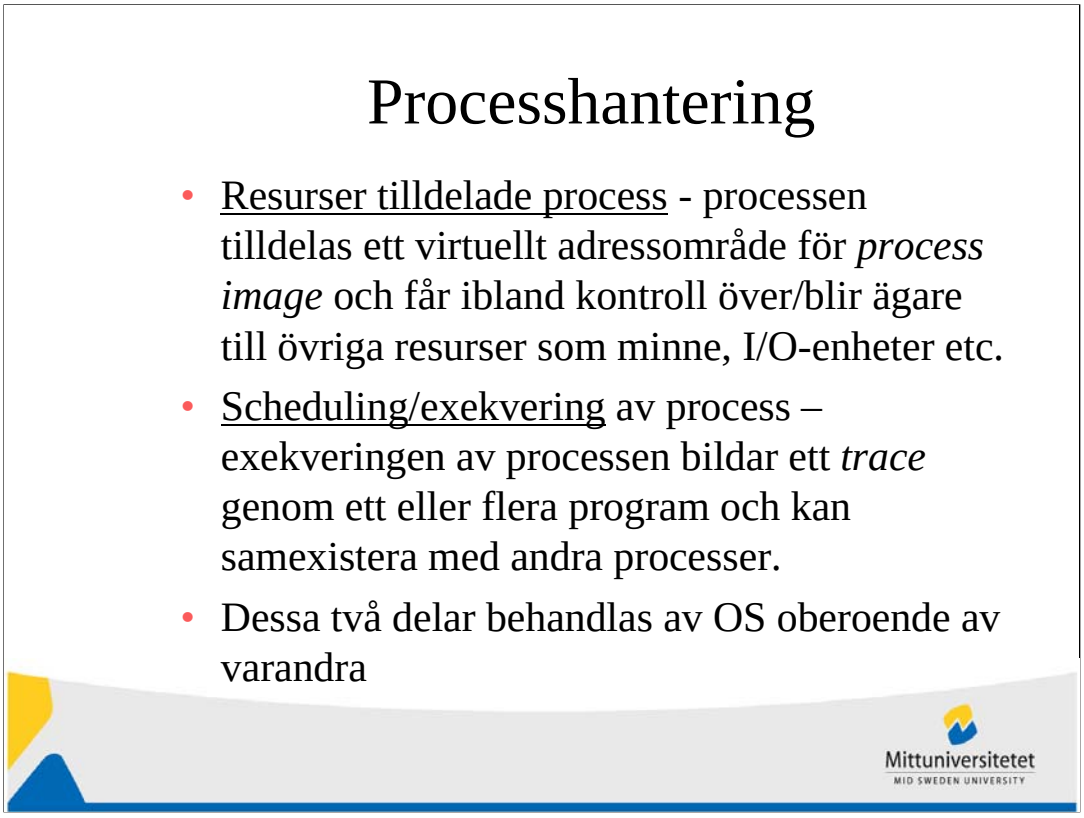
Trådar, SMP, och Mikrokernel

Mittuniversitetet
MID SWEDEN UNIVERSITY

Kapitel 4 (sid. 157-197)

Förra kapitlet behandlade processen. Vi ska nu gå djupare in i processen och titta på en möjlighet som moderna OS har, att dela upp en process i flera exekverbara trådar (threads).

Tanken är att när en process avbryts av t ex interaktivitet med användaren (inmatning) så finns det andra delar av processen, som inte är direkt beroende av detta, som skulle kunna fortsätta att exekvera. En tråd väntar, men andra i samma process kan fortsätta.



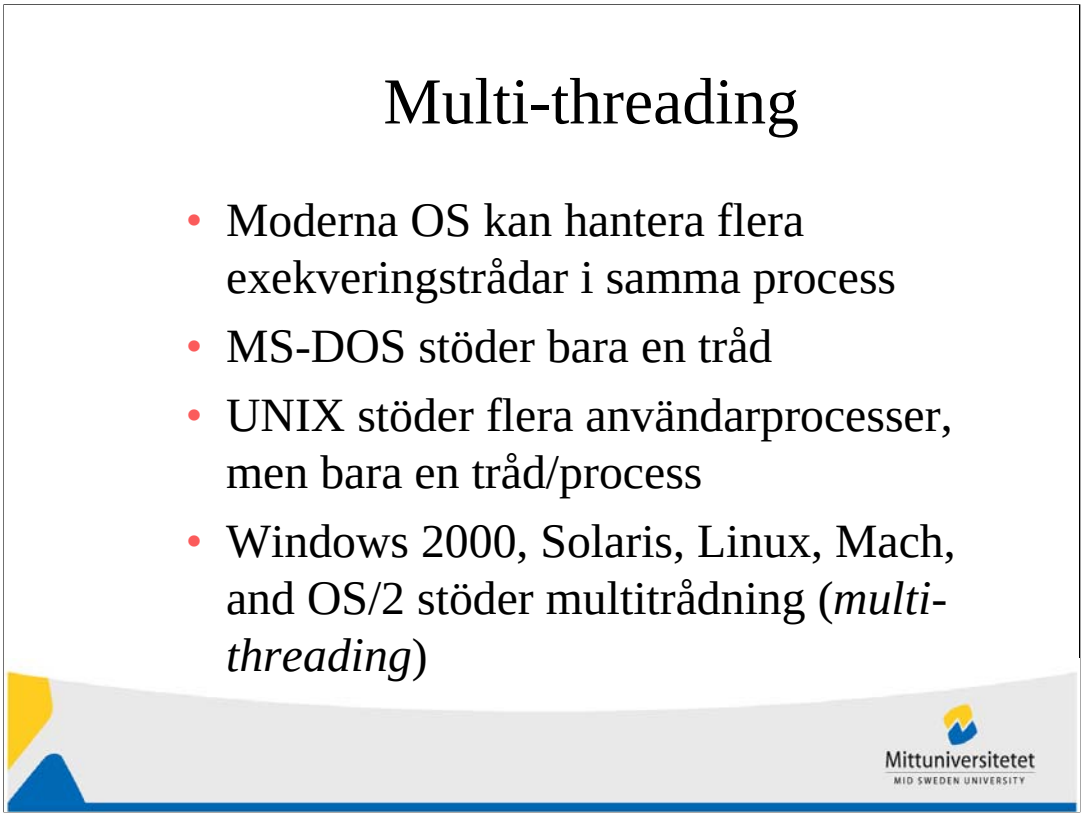
Hantering av processer

Bild 2-3: Visar hur processhanteringen kan delas upp i två olika delar, den ena som har med resurstilldelning att göra och den andra allt kring schedulering/exekvering.

Processhantering

- För att skilja på vilken hantering vi pratar om benämner vi processen
- Tråd (*Thread*) när vi diskuterar exekvering. Exekveringstråd
- Process eller *Task* beträffande resurshantering

Mittuniversitetet
MID SWEDEN UNIVERSITY



Threading - multithreading (multitrådning)

Bild 4-6: Observera att tråden beskriver det exekveringsspår som en process/tråd lämnar efter sig.

Fördel med multi-threading

- Fördelen med att ha flera trådar i samma process är att flera trådar kan exekvera ”samtidigt”.
- Om en tråd blir avbruten stannar inte hela processen, utan en annan tråd kan fortsätta.
- Olika kombinationer av processer/trådar, se nästa bild.

Mittuniversitetet
MID SWEDEN UNIVERSITY

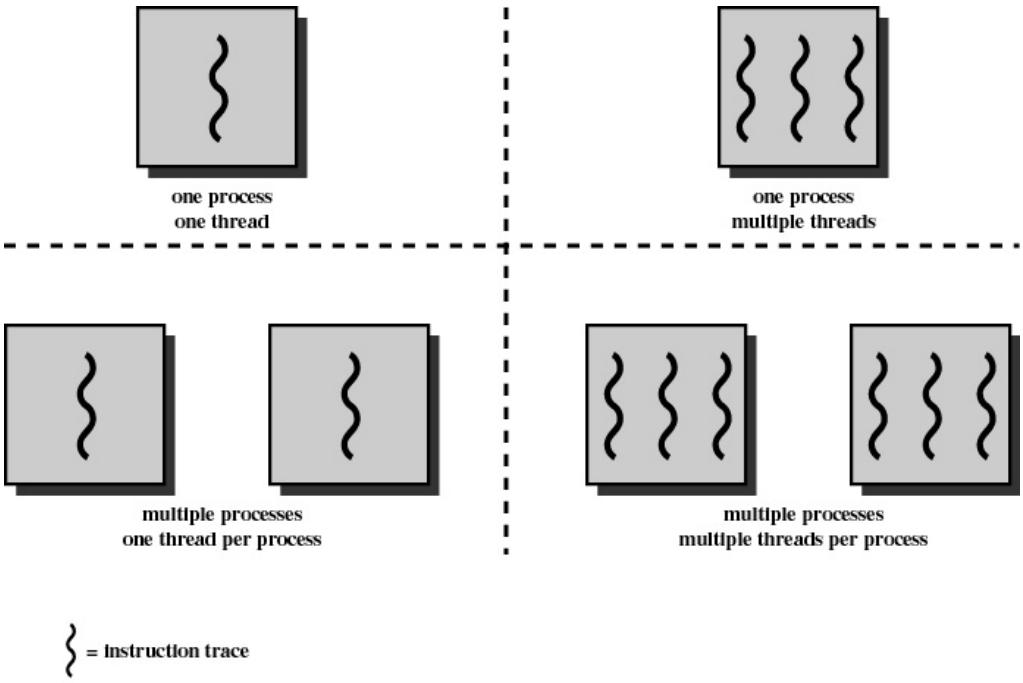


Figure 4.1 Threads and Processes [ANDE97]

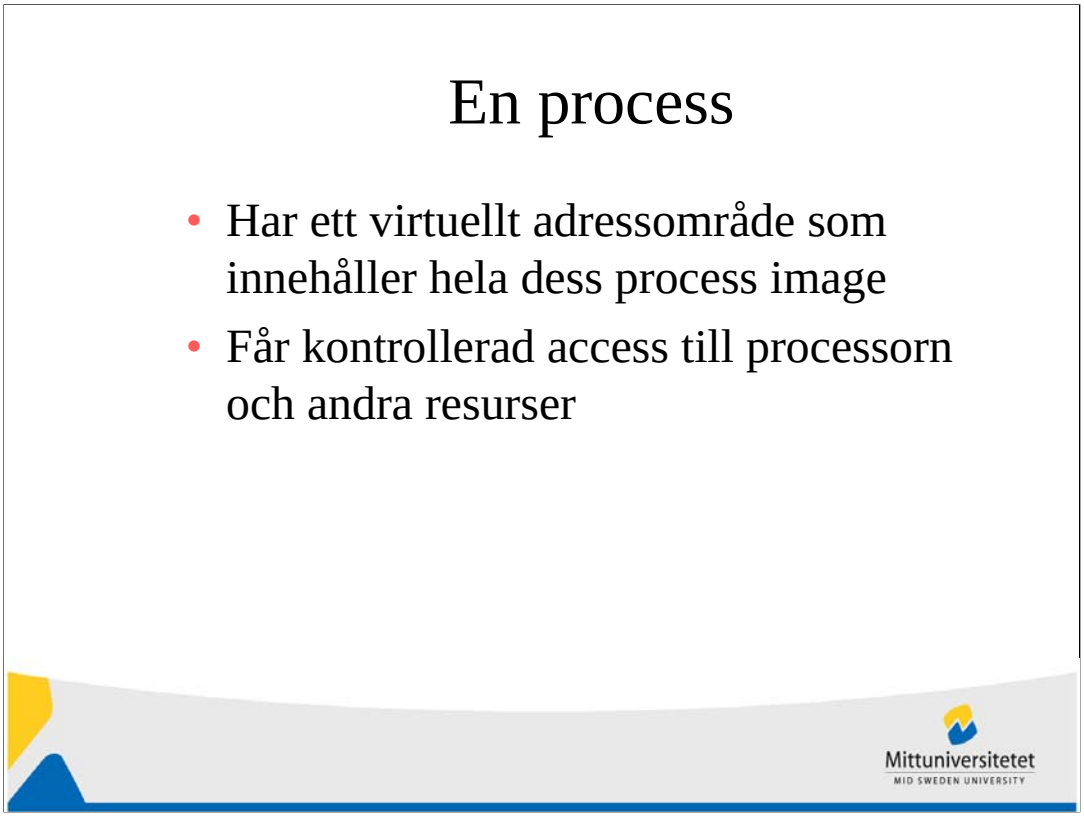


Bild 7-12: Visar skillnaden mellan process och tråd. *Bild 10:* Varje tråd måste ha en egen stack och ett trådkontrollblock, men de delar adressområde och processkontrollblock med alla trådar i processen

En tråd har

- ett exekveringstillstånd (running, ready osv.)
- ett sparat context när den inte är running
- en exekveringsstack
- trådspecifik minnesarea för lokala variabler
- Viss tillgång till ”sin” process minne och resurser
  - Alla processens trådar delar detta

Mittuniversitetet
MID SWEDEN UNIVERSITY

- Enligt tidigare så har varje process ett processkontrollblock, ett virtuellt adressområde och en stack (user/kernel)
- Vid multitrådning måste varje tråd ha egen stack, eftersom de exekverar oberoende. Dessutom tillkommer trådspecifik information i ett trådkontrollblock/tråd.
- Samtliga processens trådar delar på adressområde och processkontrollblock. Se nästa bild.

Mittuniversitetet
MID SWEDEN UNIVERSITY

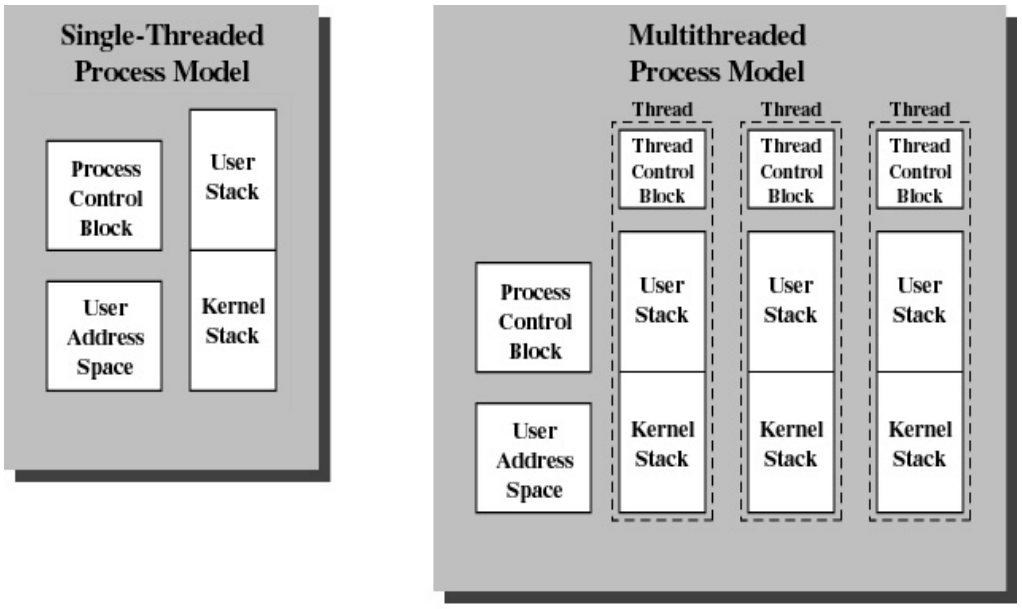


Figure 4.2 Single Threaded and Multithreaded Process Models

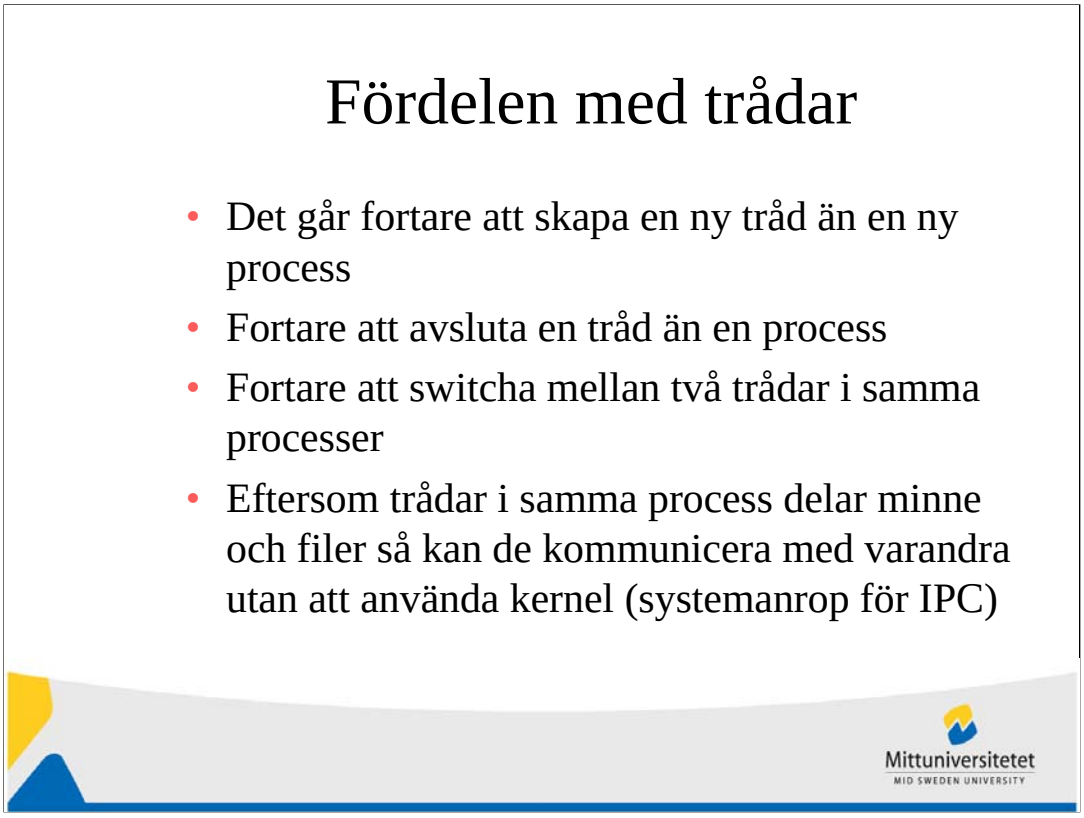
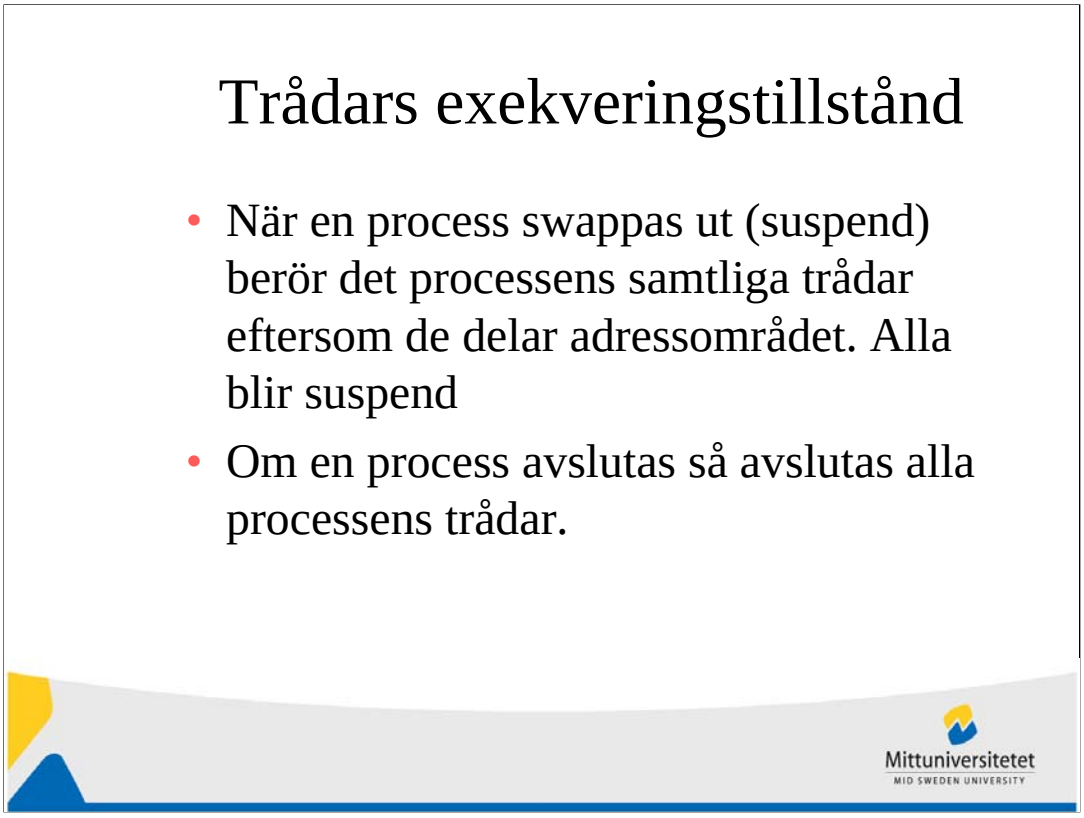


Bild 11: sammanfattar fördelarna.

Användande av trådar i enprocess och multiprocess-system

- En tråd kan arbeta i förgrunden, interaktivt mot användare, medan en annan arbetar i bakgrunden med andra uppgifter
- Asynkrona processer, en tråd kan hantera ex vis periodisk backup
- Hastigheten kan öka, genom att flera trådar i samma process kan exekvera samtidigt
- Enklare att bygga modulära program om varje tråd hantera en begränsad uppgift

Mittuniversitetet
MID SWEDEN UNIVERSITY



Trådars exekveringstillstånd

Bild 13-14: Till viss del liknar det processens, men det är vissa viktiga skillnader att tänka på:

• • När en process swappas ut (delar/hela processen flyttas ut från primär till sekundärminne) så berör det samtliga trådar, eftersom alla delar samma adressområde.

• • Alla trådar tillhörande samma process måste avslutas när processen avslutas, annars skulle de bli hängande i luften.

Operationer Som Påverkar En Tråds Exekveringstillstånd

- *Spawn* (Skapa)
  - skapar en ny tråd
- *Block*
- *Unblock*
- *Finish*
  - Frigör register och stackar

Mittuniversitetet
MID SWEDEN UNIVERSITY

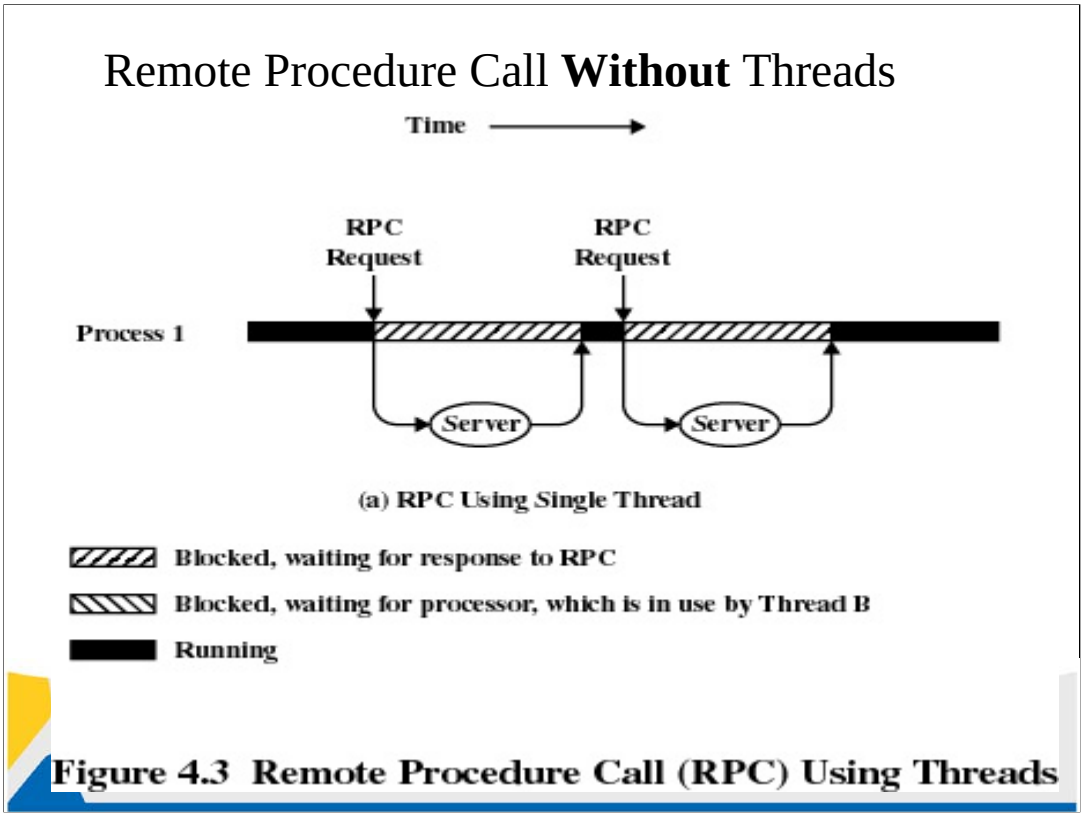


Bild 15-16: Visar en jämförelse mellan hur samma två anrop till en server skulle se ut, utan och med trådar i processen. Man ser tydligt hur mycket effektivare det blir när tråd B kan fortsätta att exekvera medan tråd A väntar på svar.

Se också Figure 4.4 i boken.

Läs också exemplet om Adobe PageMaker

Remote Procedure Call Using Threads

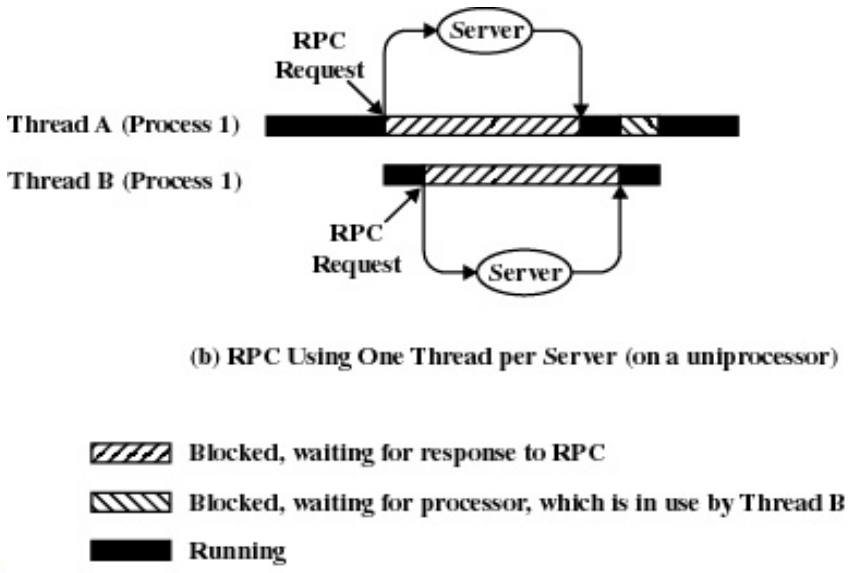


Figure 4.3 Remote Procedure Call (RPC) Using Threads

User-level/Kernel-level threads

Ni trodde väl inte att det var så enkelt…

Bild 17-22: Observera först skillnaden mellan ULT och KLT.

ULT först.

När ni försöker sätta er in i Figure 4.7 med hjälp av texten på sidan 163, så saknas det en liten sak: Mellan 2 och 3 så har process B fått börja exekvera igen (Annars hade ju inte tråd 2 kunnat komma vidare till att behöva data från tråd 1). Det är alltså två trådar som båda tillhör process B och de exekveras i ett rent ULT-system.

Längst ner på sidan 163 ff tas fördelar med ULT över KLT upp:

• Mindre overhead vid trådswitchning vid ULT

• Scheduleringen av trådarna kan bestämmas av applikationen, eftersom OS inte ser trådarna. Applikationen kan prioritera vissa trådar.

• ULT kan köras på alla OS eftersom *threads library* ligger på applikationsnivå.

Nackdelarna är:

• Om en av processens trådar gör ett systemanrop och blockeras, så blockeras alla trådar i processen

• Ren ULT kan vid multitrådning inte utnyttja fördelarna av ett multiprocessorsystem. Kernel ser ju inte trådarna och kan därför inte låta olika

trådar från samma process exekvera på olika processorer.

Man kan komma runt nackdelarna genom att skriva applikationen med många processer, men då missar man fördelen med den enklare switchningen av trådar.

KLT

I ett kernel-level thread system så sköts all trådhantering av kerneln. Det behövs ingen trådmanagement i applikationen, OS erbjuder ett API för hantering av trådarna. Schedulering sker av kernel helt på trådbasis.

Fördelarna är att kerneln kan schedulera trådar från samma process till flera processorer och om en tråd i en process blir blocked så kan någon annan tråd i processen få fortsätta. En annan fördel är att själva kerneln kan vara trådad.

Nackdelen är att en switchning mellan två trådar inom samma process, som var effektivt i ULT, nu kräver ett mode switch till kerneln. Det är ju kernel som hanterar trådarna i KLT.

Table 4.1 sidan 166 visar skillnaderna i tid, för en operation mellan ett rent ULT, ett rent KLT och en process.

Som vanligt så blir det fel hur man än gör och en kombination av ULT och KLT, som i Solaris, är en möjlighet.

Table 4.2 sidan 167 visar olika möjliga relationer mellan processer och trådar. M:1 som är det vanligaste i ett modernt OS och 1:M speciellt intressant för distribuerade system.

Nu ska vi hoppa lite. Kapitel 4.2 Symmetric Multiprocessing och 4.3 Microkernels är intressanta men är kursiva (ni behöver inte läsa dom). *Bild 23-27 kursivt*

Vi kör vidare med

Kernel-Level Threads

- Kernel hanterar context för processen och processens trådar
- Applikationen behöver inte bekymra sig om trådhantering
- Scheduling sker på trådnivå
- Windows 2000, Linux och OS/2 är exempel på operativsystem som hanterar kernel-level threads

Mittuniversitetet
MID SWEDEN UNIVERSITY

Kombination av user/kernel-level threads

- Exempel Solaris
- Trådarna skapande, schedulering och synkronisering sker på user-nivå
- Programmeraren kan bestämma hur många kerneltrådar som som ska användas, för att optimera prestanda

Mittuniversitetet
MID SWEDEN UNIVERSITY

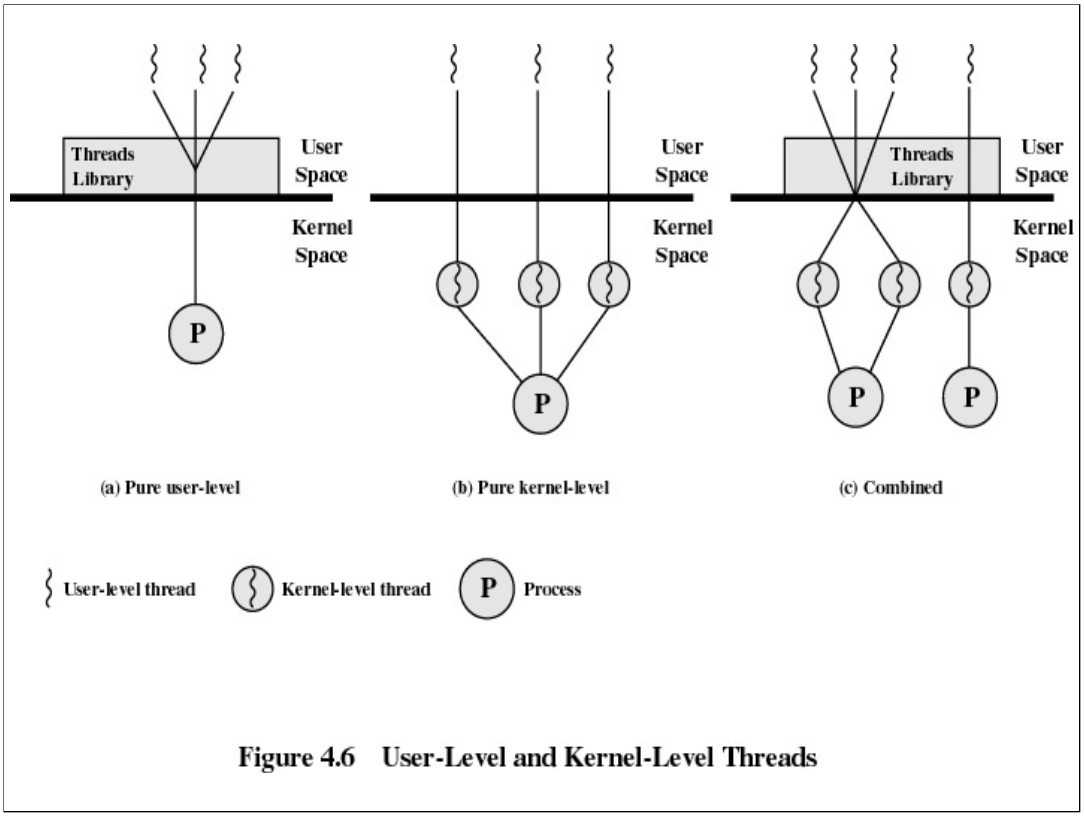


Figure 4.6 User-Level and Kernel-Level Threads

Olika Möjliga Kopplingar Mellan Trådar Och Processer

Threads:Process	Description	Example Systems
1:1	Varje exekveringstråd är en unik process med ett eget adressområde och egna resurser	Traditionell UNIX
M:1	En process definierar ett adress-område och dynamiska resurser. Multipla trådar kan skapas och ekexvera i processen	Windows NT, Solaris, OS/2, OS/390, MACH

1:1 = en-till-en M:1 = många-till-en

Olika Möjliga Kopplingar Mellan Trådar Och Processer

Threads:Process	Description	Example Systems
1:M	En tråd kan flytta från en process till en annan. En tråd kan fås att migrera mellan olika system.	Ra (Clouds), Emerald
M:M	Kombinerar delar av M:1 och 1:M	TRIX

Typer Av Multiprocessing

- Single Instruction Single Data (SISD)
  - en processor exekverar en ström av instruktioner, som opererar på data lagrat i ett minne
- Single Instruction Multiple Data (SIMD)
  - en instruktion exekverar samtidigt på olika data i olika processorer

Typer Av Multiprocessing

- Multiple Instruction Single Data (MISD)
  - en ström av instruktioner skickas till ett antal processorer, som var och en exekverar olika sekvens av instruktioner.
    (aldrig implementerat)
- Multiple Instruction Multiple Data (MIMD)
  - ett antal processorer exekverar samtidigt olika sekvenser av instruktioner på olika data

Mittuniversitetet
MID SWEDEN UNIVERSITY

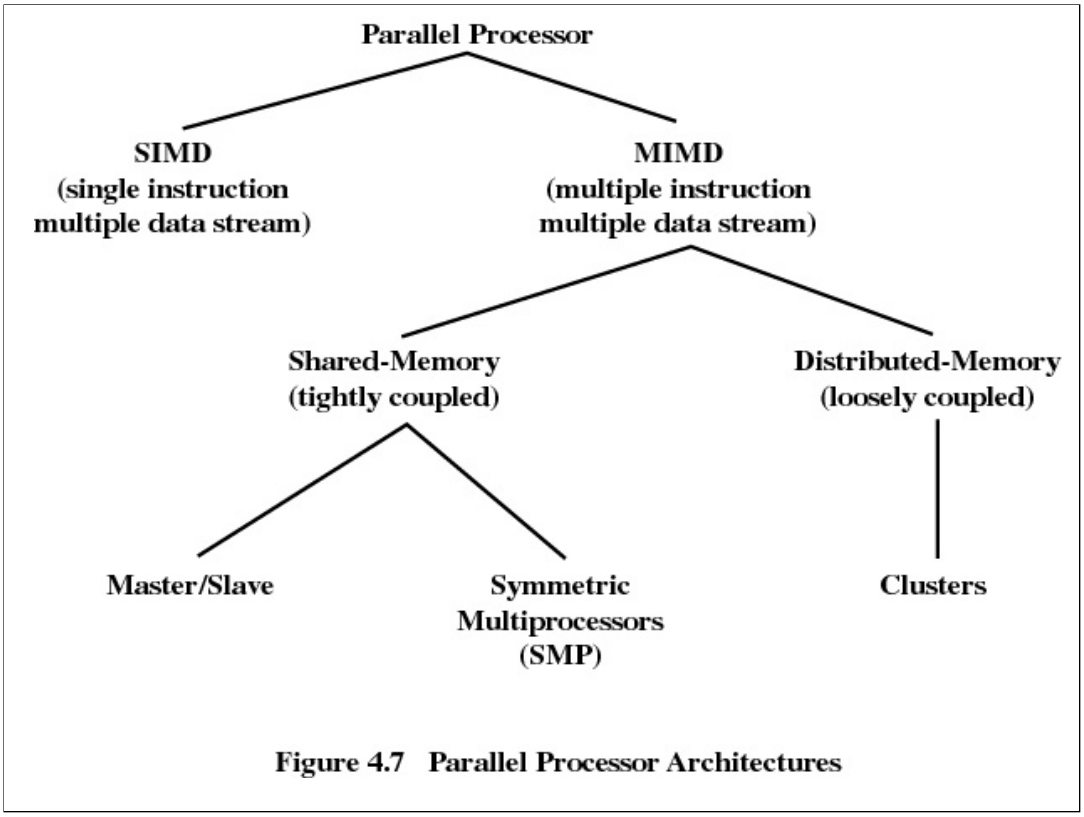


Figure 4.7 Parallel Processor Architectures

Symmetrisk Multiprocessing

- Kernel kan exekveras i någon av processorerna
- Varje processor hanterar schedulering från en pool av tillgängliga processer eller trådar

Mittuniversitetet
MID SWEDEN UNIVERSITY

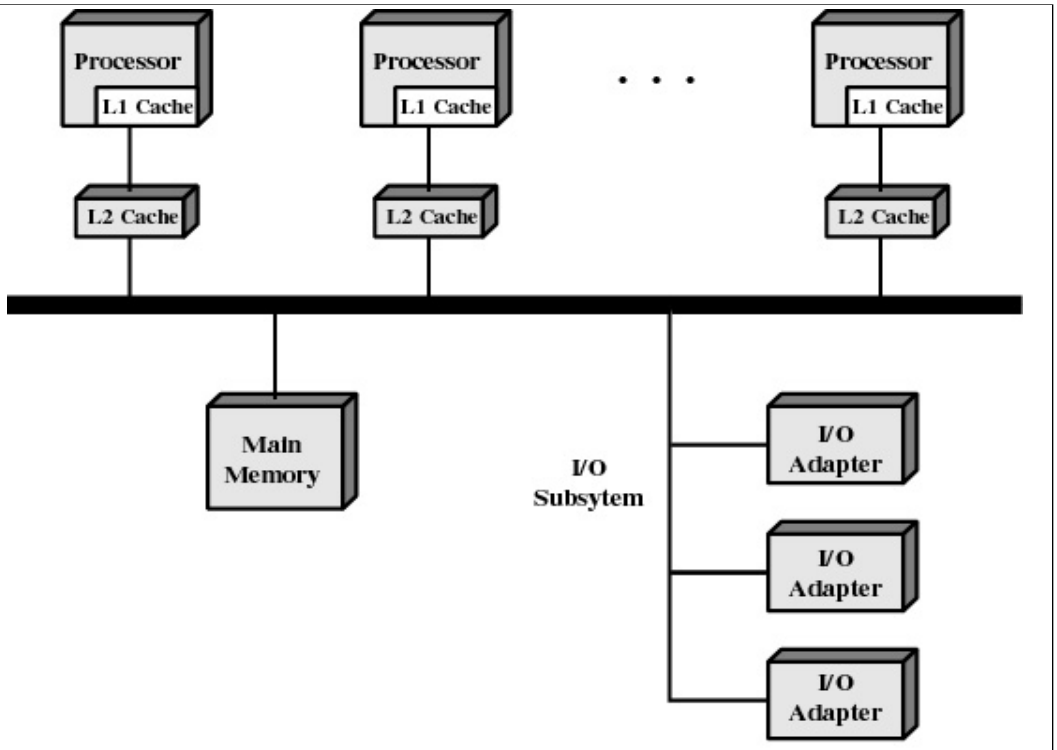


Figure 4.9 Symmetric Multiprocessor Organization

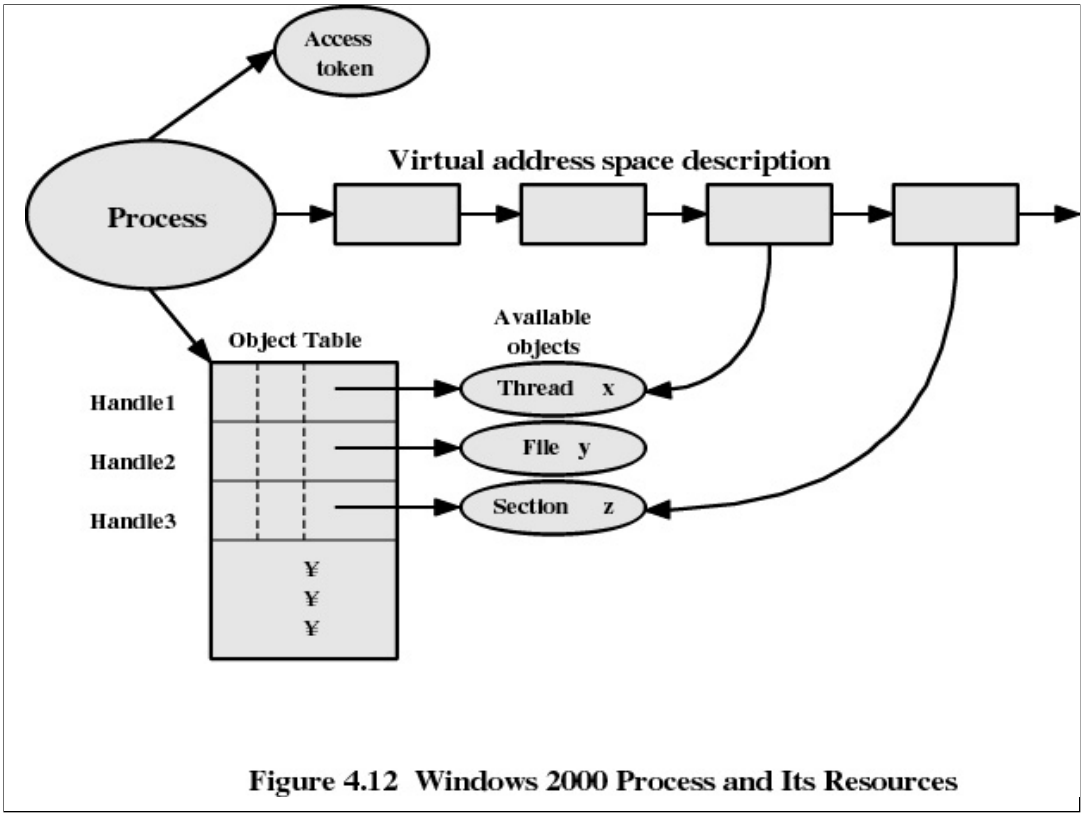
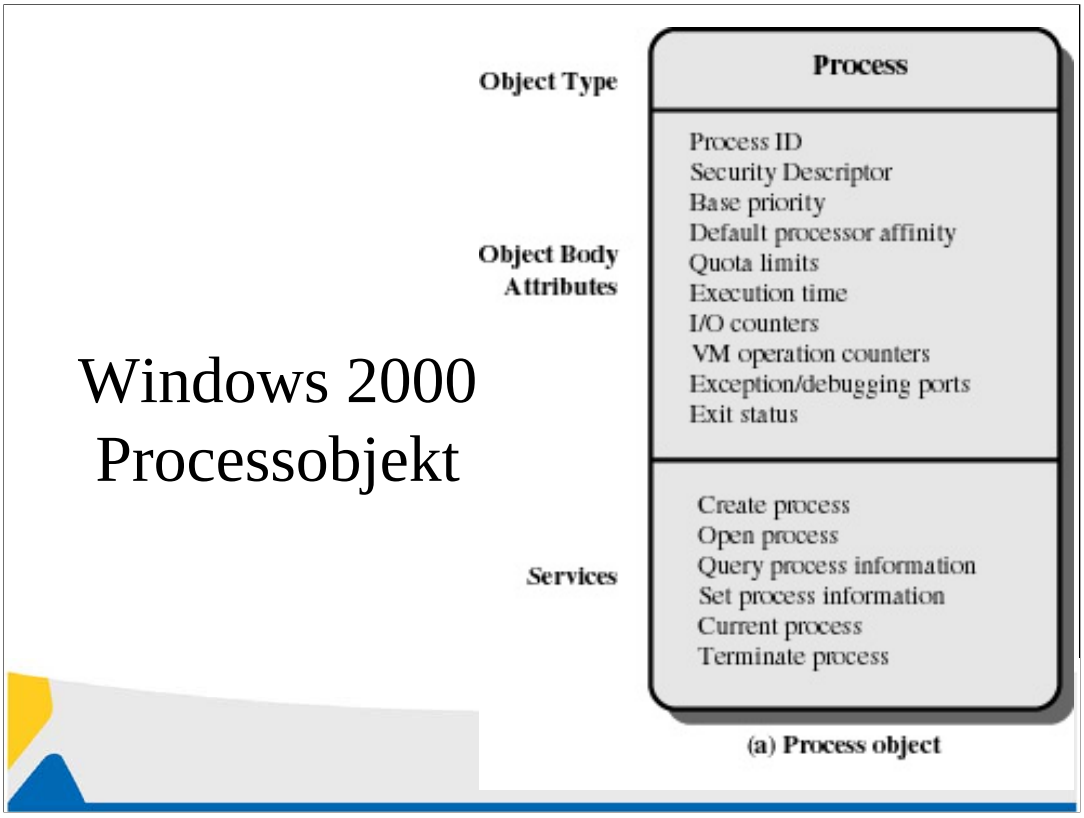


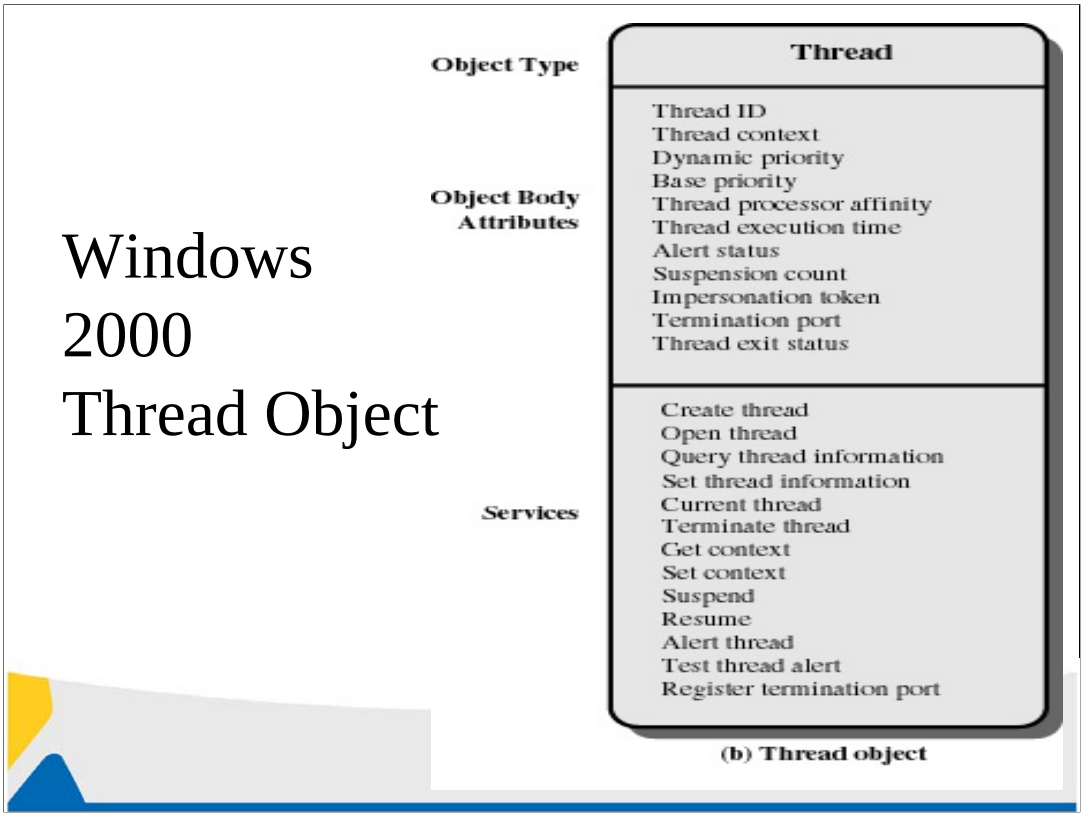
Figure 4.12 Windows 2000 Process and Its Resources



Processobjekt

W2K använder, som vi sa tidigare, två typer av processrelaterade objekt: process och tråd.

Bild 29: Visar processobjektet. Som ett vanligt objekt har det ett antal attribut och ett antal services (metoder heter det på Svenska va?). För att få en metod utförd så måste man skicka en begäran. Se även Table 4.3 sidan 185 som förklarar attributen.



Trådobjektet

Bild 30: tillsammans med Table 4.4 visar motsvarande för trådobjektet. Minst en tråd måste finnas som sedan kan skapa flera trådar.

Något attribut i trådobjektet liknar det i processobjektet. Tråden har då ärvt detta från processobjektet den tillhör.

Multithreading

W2K stöder concurrency för trådar och alltså för processer. Dessutom kan trådar i samma process vara samtidiga i ett multiprocessorsystem.

Trådar i samma process kan kommunicera via den gemensamma adressarean och har tillgång till gemensamma resurser. Trådar i olika processer kan kommunicera via speciellt hanterat delat minne.

Den objektorienterade multitrådade miljö som W2K erbjuder passar bra i serverapplikationer. Som vi har tittat på i Datakommunikationskursen så skapar en begäran från en klient en ny processortråd som hanterar denna begäran

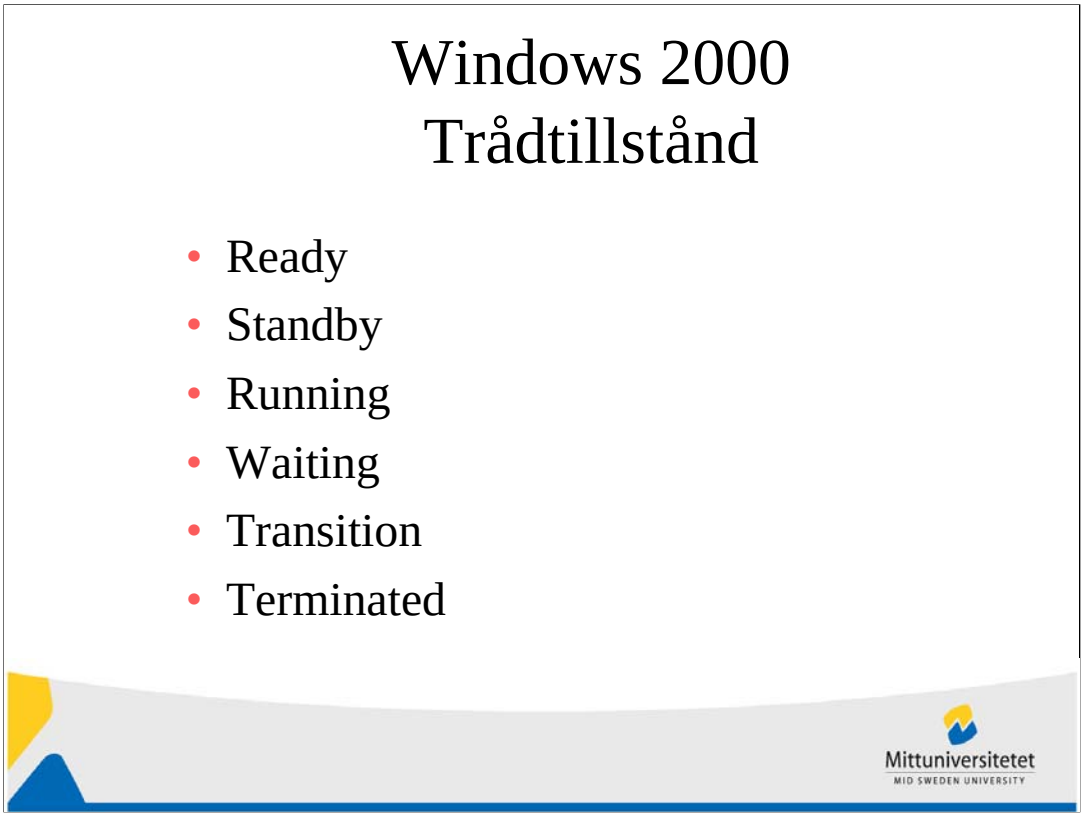


Bild 31-32: Visar tillstånden för trådar i W2K

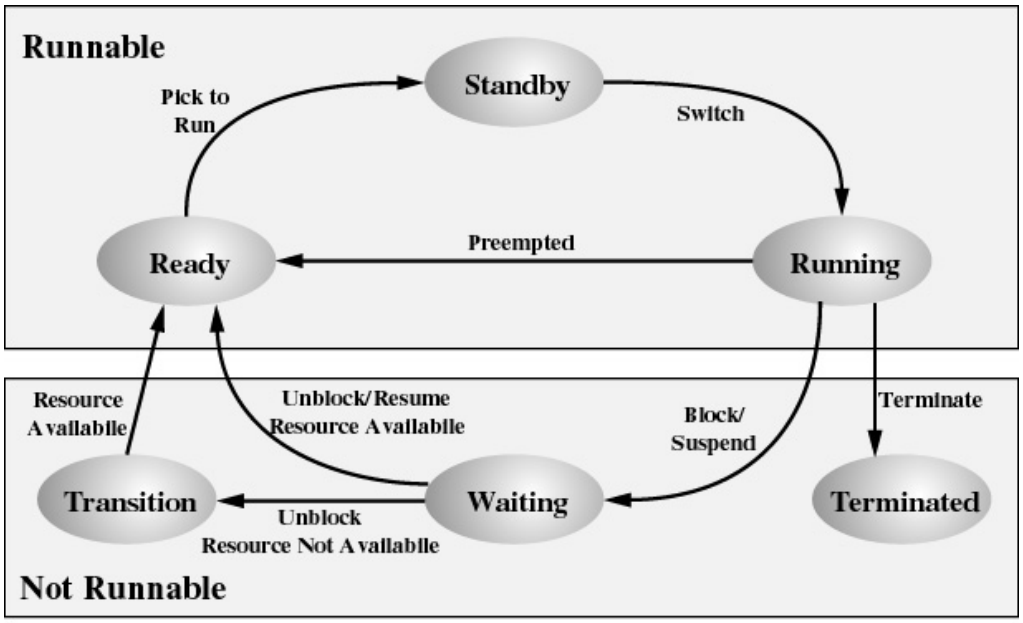


Figure 4.14 Windows 2000 Thread States

• **Ready**: *microkernelns dispatcher* plockar trådarna till exekvering i prioritetsordning.

• **Standby**: en körklar tråd kan avbryta exekveringen av en tråd med lägre prioritet. Annars väntar den till den andra trådens *time-slice* är slut. Detta är en kö till en processor.

• **Running**: som vanligt

• **Waiting**: av tre orsaker

o *blocked* p g a I/O,

o väntar på synkronisering med annan tråd

o en applikation har blockerat tråden

• **Transaction**: Motsvarar processtillståndet *Block/Suspend* sidan 121. Någon del av trådens resurser är swappad.

• **Terminated**: en tråd kan avslutas av sig själv, av annan tråd, eller av att tillhörande process avslutas. Efter garbagecollecting är tråden borta. Men systemet kan spara undan trådar som den tror kommer att behövas igen.

Support for OS Subsystem

Titta igen på sidan 85 (kanske kopiera den i A3 och sätt den ovanför sängen). W2K är ju gjord för att kunna hantera flera olika miljöer även mjukvarumässigt. Därför finns ”Environment Subsystem” som översätter OSIX, OS/2 och Win32-kommandon till generella W2K-kommandon och emulerar dessa OS.

Bokens författare skriver att det är komplicerat, viket låter illavarslande. En enkel sammanfattning gör han:

En applikation vill skapa en ny process.

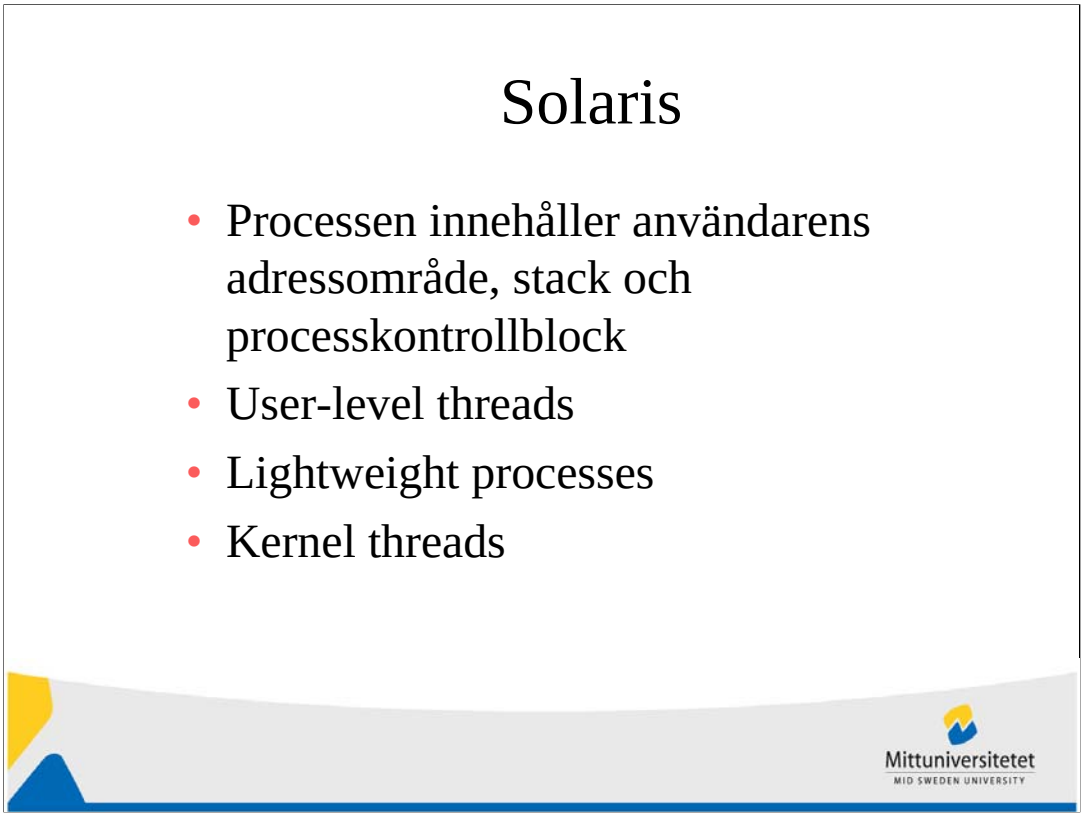
• Begäran är ju riktad till det OS som applikationen är skriven för, så motsvarande subsystem får hantera denna begäran.

• Subsystemet skickar, en omformulerad, begäran till W2K Executive.

• En process skapas av Executive och ett handtag (*handle*) lämnas till subsystemet.

• En tillhörande tråd skapas automatiskt endast för Win32 och OS/2. Även handtag för tråden skickas tillbaka.

• Vid 16-bitars Windows och POSIX, som inte stöder trådar, skapas en tråd men inget handtag tillbaka.



4.5 Solaris Thread and SMP management

Solaris är lite annorlunda uppbyggt. Det har samma process-trådrelation, M:1, men använder också ”*Lightweight processes*” (LWP)

Bild 33-34 Visar hur varje kerneltråd har en LWP som mappar en eller flera användartrådar. Exemplet visar flexibiliteten med systemat.

• Process 1 motsvarar en traditionell UNIX-process. En process - en tråd. Om en process inte behöver ha flera trådar som exekverar samtidigt, så är detta OK.

• Process 2 jobbar i ren ULT. Flera användartrådar exekverar genom en kerneltråd. Lämpligt om applikationen består av flera trådar, men de behöver inte exekvera samtidigt.

• Process 3 multipla trådar är mappade, via LWP, till ett färre antal kerneltrådar. Här kan applikationen bestämma graden av samtidighet genom att variera mappningen.

• Process 4 fast mappning en-till-en mellan tråd-LWT-kerneltråd. Detta ger full samtidighet ända till applikationen.

• Process 5 är en kombination av allt samt en direkt bindning av en tråd till en specifik processor.

I systemet finns naturligtvis flera trådar, för systemanrop, som inte är bundna till någon LWP.

Med denna uppbyggnad, kombination av ULT och KLT, fås stor flexibilitet. Den som skriver applikationen kan bestämma hur systemet ska hantera processen.

Resten av kapitlet, utom sammanfattningen, är kursiv. Även *Bild 35-40*

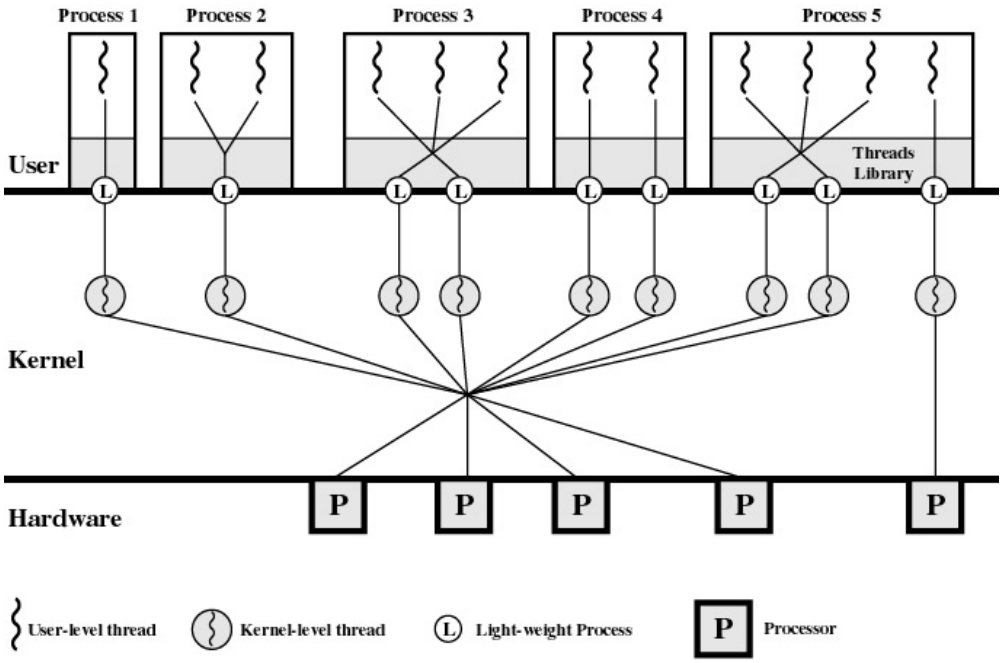


Figure 4.15 Solaris Multithreaded Architecture Example

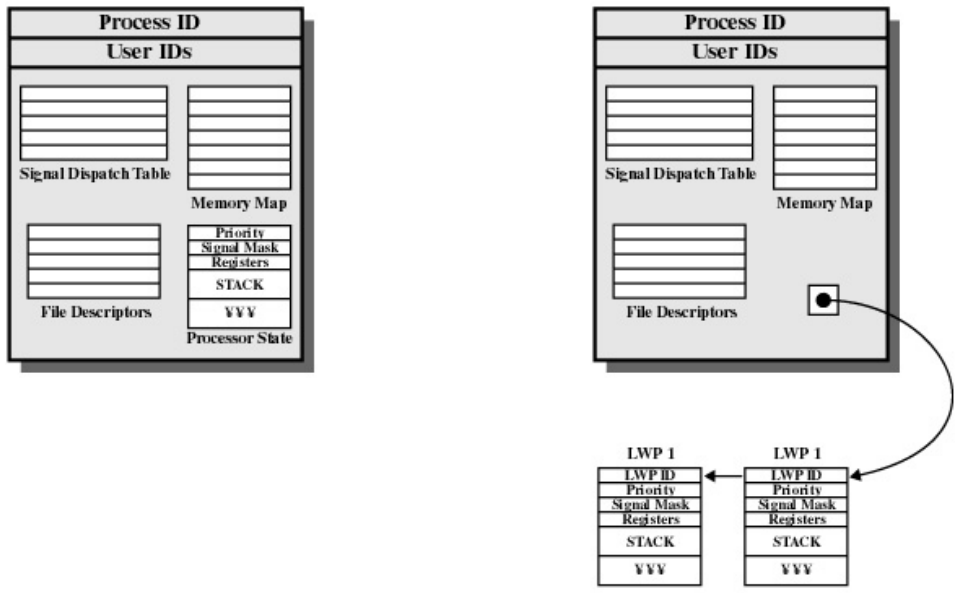


Figure 4.16 Process Structure in Traditional UNIX and Solaris [LEWI96]

Solaris exekvering Av Trådar

- Synchronization
- Suspension
- Preemption
- Yielding

Mittuniversitetet
MID SWEDEN UNIVERSITY

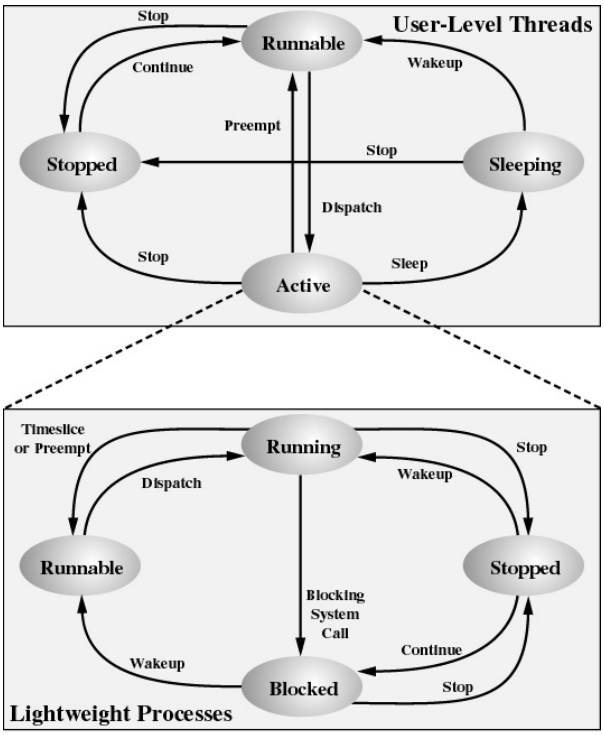


Figure 4.17 Solaris User-Level Thread and LWP States

Linux Process

- State
- Scheduling information
- Identifiers
- Interprocess communication
- Links
- Times and timers
- File system
- Virtual memory
- Processor-specific context

Mittuniversitetet
MID SWEDEN UNIVERSITY

Linux Processtillstånd

- Running
- Interruptable
- Uninterruptable
- Stopped
- Zombie

Mittuniversitetet
MID SWEDEN UNIVERSITY

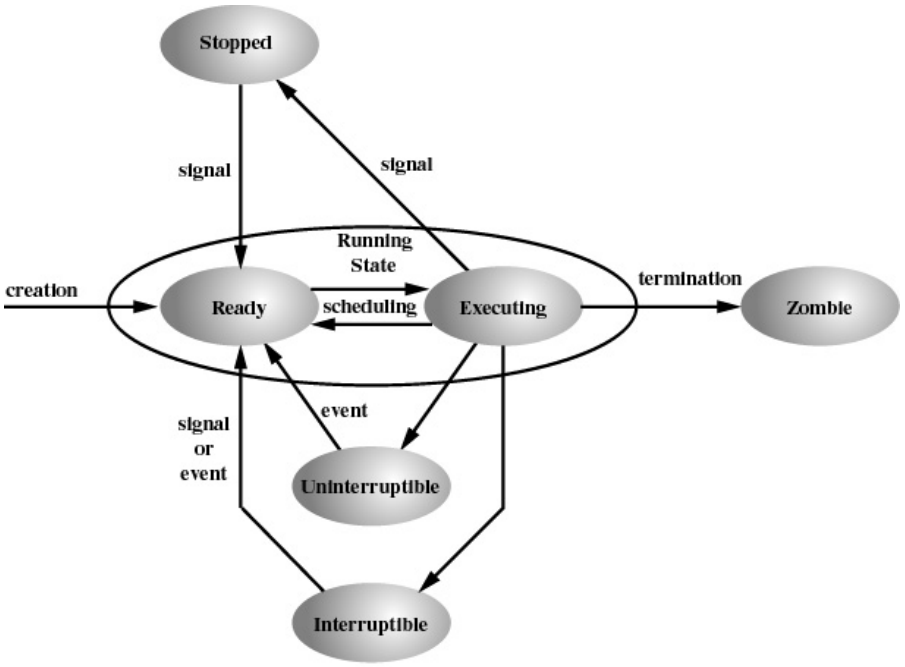


Figure 4.18 Linux Process/Thread Model