WEEK 3

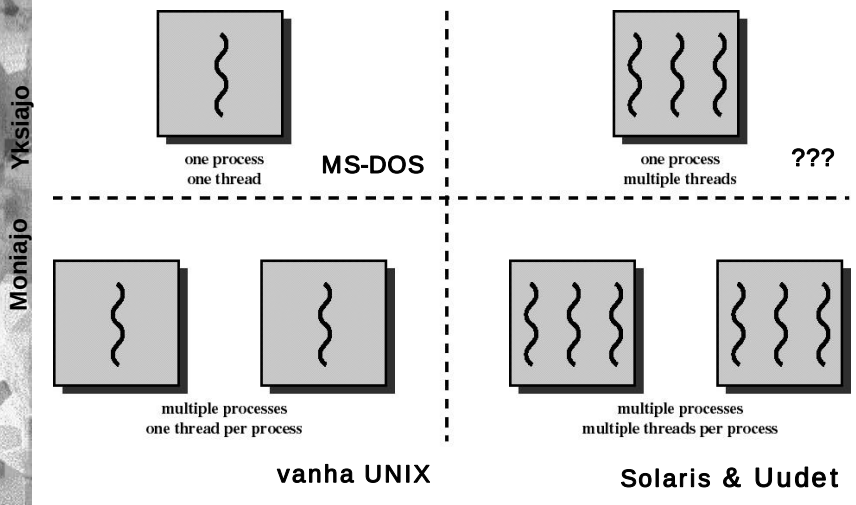
Threads

Stallings, Chapter 4

Also: Chapter

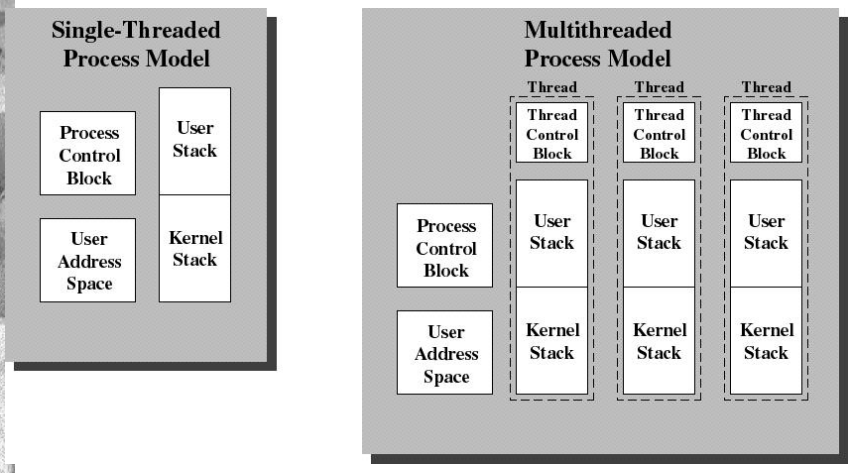
1

Processes and threads



2

Threads



Own TCB,
stack, state, storage location for registers

3

Why threads?

- Faster to create one thread than a whole process
- Dispatching threads of one process faster than dispatching processes
- When one thread waits, another might be able to execute
- Easier to finish a thread than a process
- Easier programming models are possible

4

Why to use threads?

- ‘ Efficient resource sharing by thread of one process
- ’ Easy communication between threads of one process: shared data area.

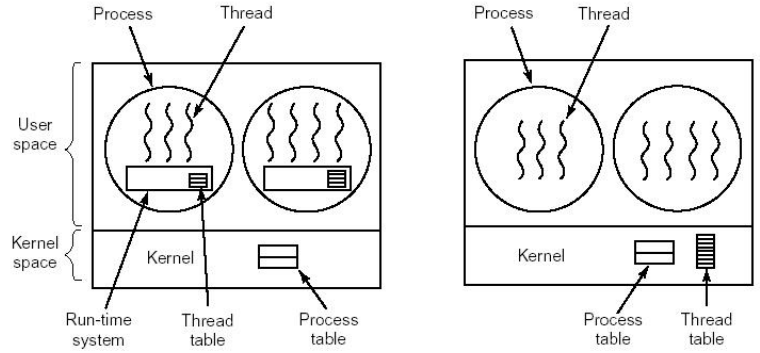
BUT

- Mutual exclusion and syncronisation are fully programmer’s responsibility (no support from OS or anything else)

ð RIO-kurssi

5

User-level vs kernel-level threads



(a) A user-level threads package.

(b) A threads package managed by the kernel.

Tan01 kuva 2-13

6

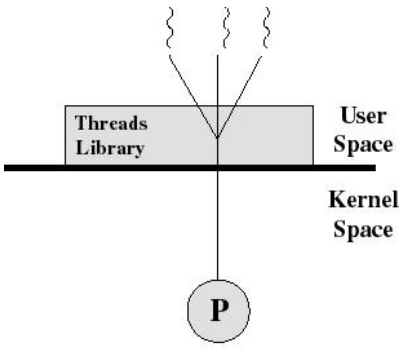
KÄYTTÄJÄTASON SÄIKEET

ULT, User Level Threads

7

User-level threads

- Kernel (or OS) is not aware of threads, it controls and dispatches only processes
- User process uses a thread library to control threads
- User porocess must dispatch threads itself
  - Programmer's task
  - Not done by the kernel
  - Not based on interrupts



8

POSIX threads (pthreads)

- pthread_create()
  - parametrina funktio, josta suoritus alkaa
- pthread_exit()
  - lopeta säikeen suoritus
- pthread_join()
  - odota parametrina annetun säikeen loppumista
- Synkronointi, poissulkeminen (semaforit)
  - pthread_mutex_init() / _destroy()
  - pthread_mutex_lock() / _trylock() / _unlock()
- Ynnä muita funktioita
  - sched_yield(): luovu vapaaehtoisesti CPU:sta

9

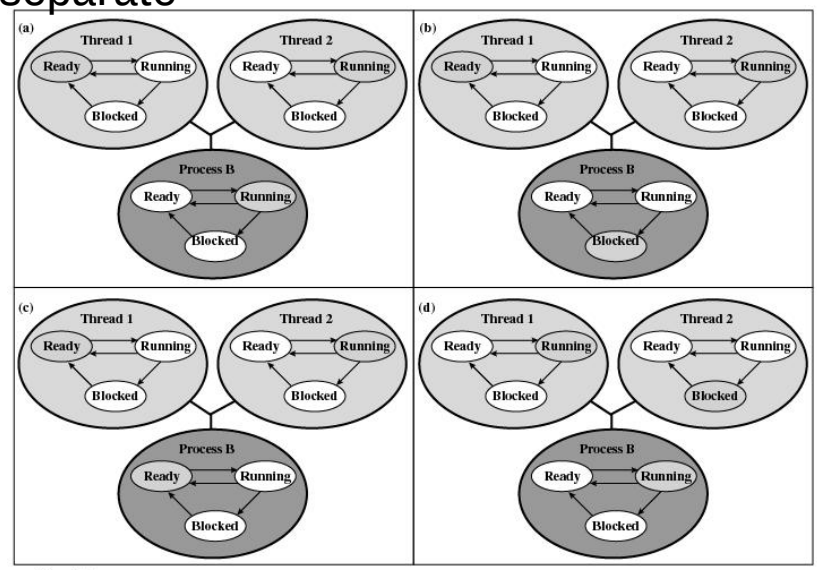
Example

```
void main() {
   pthread_t thr1, thr2;
   char *msg1 = "Hello";
   char *msg2 = "World";
   pthread_create(&thr1, pthread_attr_default,
  (void*)&print_message_function,(void*)msg1);

   pthread_create(&thr2, pthread_attr_default,
  (void*)&print_message_function,(void*)msg2);
   exit(0);
}
void print_message_function(void *ptr){
   char *message;
   message = (char *) ptr;
   printf("%s ", message);
}
```

10

User level thread's and process' states are separate



Colored state
is current state

Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

User-level threads

Advantages

- Fast dispatching
  - No mode switch
  - No interrupt
  - No process switch!
- Programmer can freely choose the scheduling mechanism

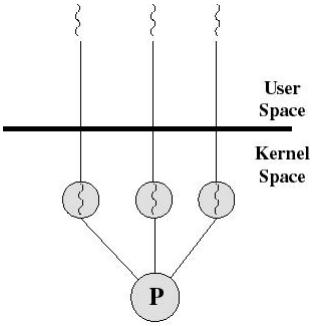
Disadvantages

- When one thread is blocked on system call, it blocks the whole process (and all other threads)
- Threads of one process cannot be executed on several processors concurrently
  - Remember: kernel dispatches only processes!

12

Kernel-level threads

- All thread control done by the kernel
  - TCB and PCB maintained by kernel
- Kernel dispatches the threads now
- No control on the user level (thus no need for user-level thread library to control execution)



13

Kernel-level threads

Advantages

- Threads of one process can be executed simultaneously on multiple processors
- If one thread is Blocked, the other threads of this process may still continue
- Often also the kernel implementation is multithreaded

Disadvantages

- Dispatching a thread has two phases:
  - Interrupt +, interrupt handling and mode switch
  - Dispatcher (and return to user mode)
- Slower than in user-level threads

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

14

Solaris – combining user-level and kernel-level threads

Ch 4.5 – 4.6 [Stal05]

15

Solaris

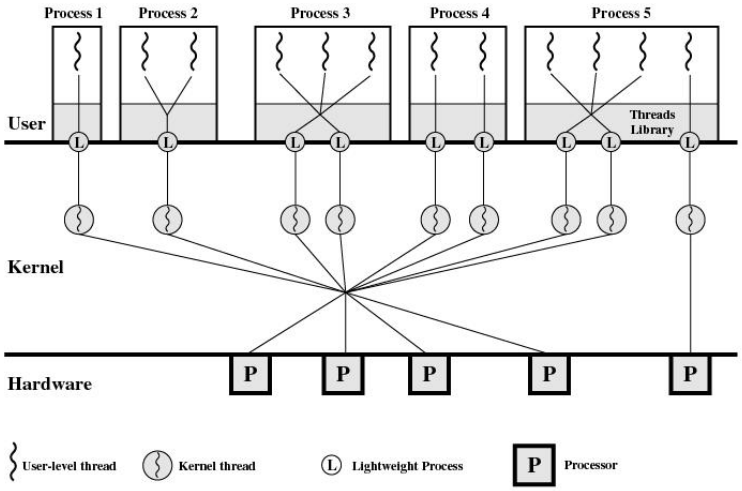


Figure 4.15 Solaris Multithreaded Architecture Example

16

Solaris: ULT state transitions

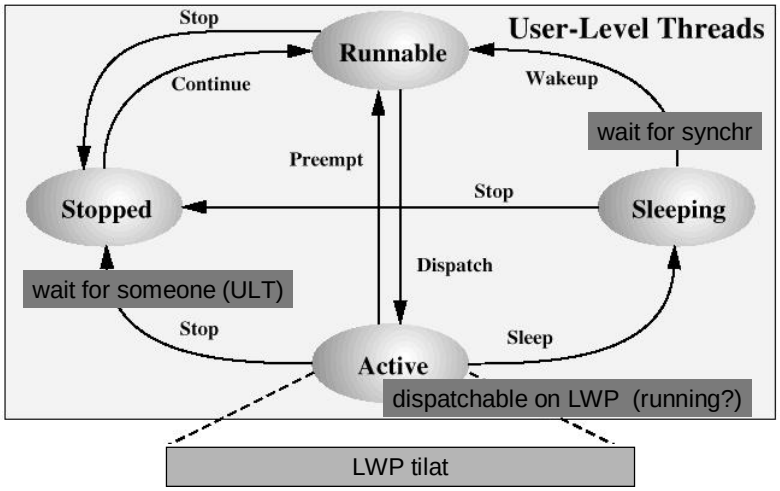


Fig 4.17 upper [Stal05]

- Binding user-level threads with kernel-level threads?

17

Solaris: LWP state transitions

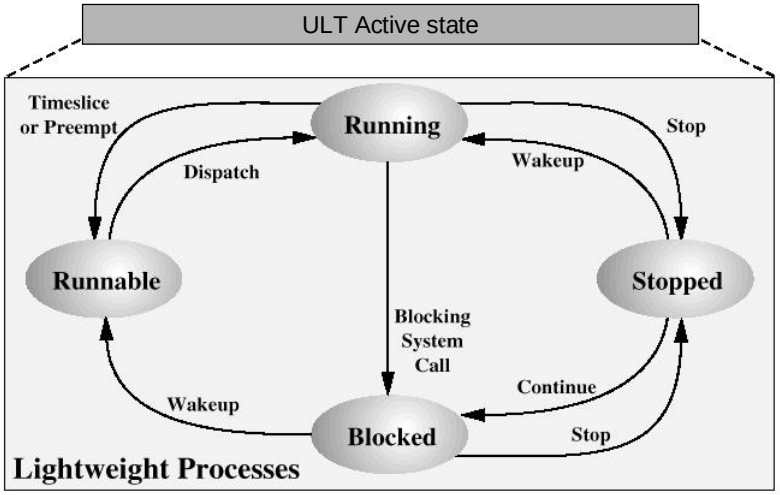
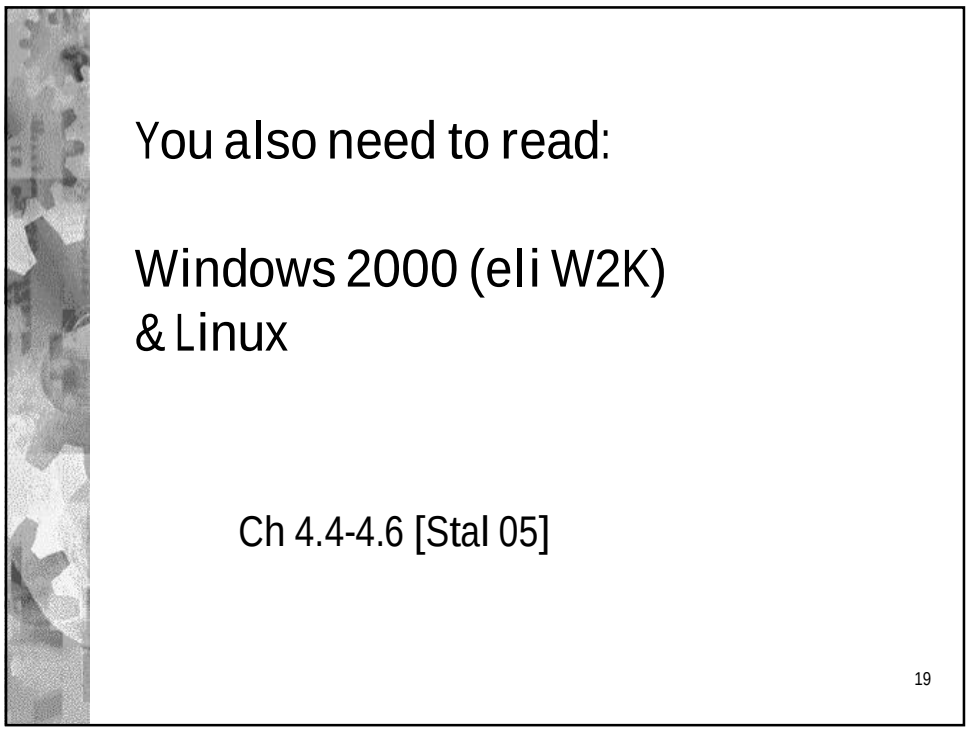
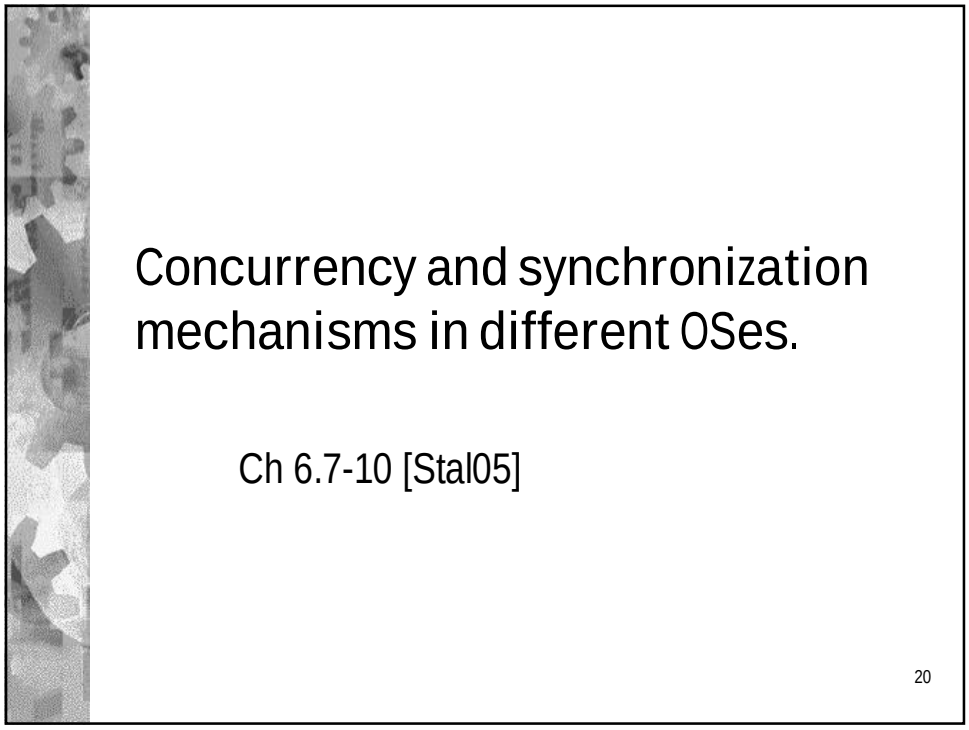


Fig 4.17 lower [Stal05]

18


You also need to read:
Windows 2000 (eli W2K)
& Linux
Ch 4.4-4.6 [Stal 05]
19


Concurrency and synchronization
mechanisms in different OSes.
Ch 6.7-10 [Stal05]
20

IPC and Unix

IPC Type	POSIX.1	SVR4	4.4BSD
Signals	•	•	•
Pipes (HDX)	•	•	•
FIFOs (named pipes)	•	•	•
Stream pipes (FDX)		•	•
Named stream pipes		•	•
Message queues		•	
Semaphores		•	
Shared memory		•	
Sockets		•	•
Streams		•	

21

Concurrency mechanisms in Linux

Ch 6.8 [Stal 05]

- As other UNIX
  - Pipes, messages, shared memory, signals, semaphores
- In kernel mode also
  - atomic *int*- and *bitmap*-operaations
  - *spinlock* to protect critical areas
  - *spinlock* variants to protect critical areas of interrupt handlers
  - *reader-writer spinlock* for multiple reader – one writer
  - *kernel semaphore*
  - *barrier* to enforce the instruction execution order (information for compiler and processor hardware), not to allow reordering

22

spinlock

- Spinlock implementation depends on the number of processors (and pre-emption)
- Uniprocessor
  - If kernel non-preemptable, no locks needed, omitted by the compiler
  - If kernel preemptable, replaced with disable/enable interrupts.
- Multiprocessor, SMP
  - Spinlock implemented as busy-wait
- NOTICE: No changes to the actual code, only to the compilation

23

Solaris

- Mutexes, semaphores and reader/writer lock have a special *nonblocking* operation (in addition to the more traditional operations)
  - Mutex_tryenter, sema_tryp, rw_tryenter
- Relate this to the solaris multilevel thread model to understand, why it is provided

24

Solaris condition variables

- Any condition (even if non atomic calculation)
  - Must be protected with dedicated mutex
  - *cv_wait(&cv, &mutex)* releases the mutex for the wait
  - Condition must be retested once more after the wait, the reacquiring of the mutex might have taken time

Many try, only one can pass

```
mutex_enter(&mutex);
        while (”complex condition”)
                cv_wait( &cv, &mutex)
…critical region …
mutex_exit(&mutex)
```

kriitt. vaihe I

kriitt. vaihe II

kriitt. vaihe III

25

Windows

Ch 6.10 [Stal05]
(+ Sect 11.4 [Tane91])

Microsoft Windows 2000

26

Table 6.7 Windows Synchronization Objects

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Event	An announcement that a system event has occurred	Thread sets the event	All released
Mutex	A mechanism that provides mutual exclusion capabilities; equivalent to a binary semaphore	Owning thread or other thread releases the mutex	One thread released
Semaphore	A counter that regulates the number of threads that can use a resource	Semaphore count drops to zero	All released
Waitable timer	A counter that records the passage of time	Set time arrives or time interval expires	All released
File change notification	A notification of any file system changes.	Change occurs in file system that matches filter criteria of this object	One thread released
Console input	A text window screen buffer (e.g., used to handle screen I/O for an MS-DOS application)	Input is available for processing	One thread released
Job	An instance of an opened file or I/O device	I/O operation completes	All released
Memory resource notification	A notification of change to a memory resource	Specified type of change occurs within physical memory	All released
Process	A program invocation, including the address space and resources required to run the program	Last thread terminates	All released
Thread	An executable entity within a process	Thread terminates	All released

Uncolored (upper rows)

Note: Colored rows correspond to objects that exist for the sole purpose of synchronization.