Operating Systems: Processes and Threads

Week 1: Lecture 2, Thu 4.9.2008

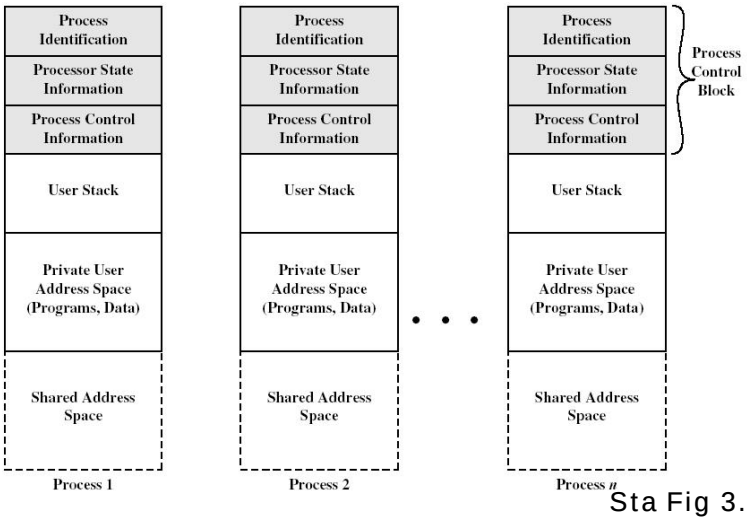
Tiina Niklander

1

Process model

2

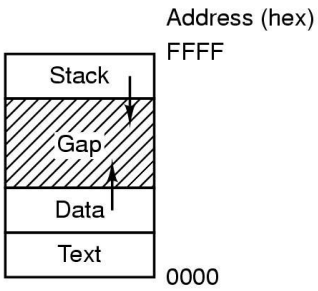
Process



Sta Fig 3.12

3

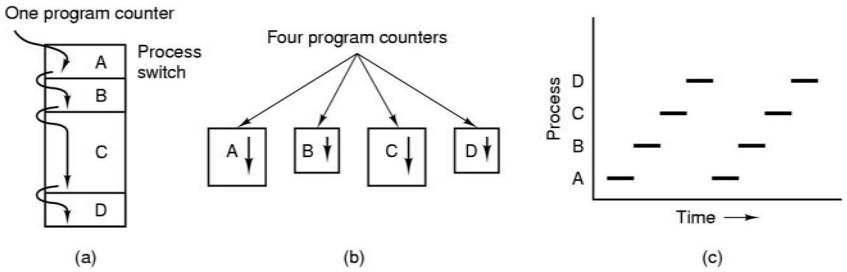
Process



- is an activity that has a program, input, output and a state.
- Some terms:
- Text/code = executable instructions
- Data = variables
- Stack = workarea
  - Parameter passing to subroutines/system calls
- Process Control Block, PCB entry in Process Table = management information

4

Process model



- One physical program counter switches between processes
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant on one CPU pseudoparallelism

5

Process Creation

Principal events that cause process creation

1. System initialization
2. Execution of a process creation system
3. User request to create a new process
4. Initiation of a batch job

6

Process Creation

Operating system does

- Create PCB
  - OS 'generates' a unique ID
- Allocate memory for the process
- Initiate PCB
- Link PCB to other structures
  - Place to Ready-queue, link to parent process, etc

7

Process Termination

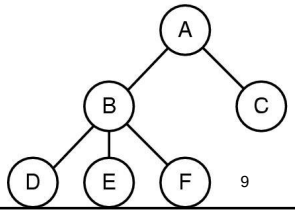
Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

8

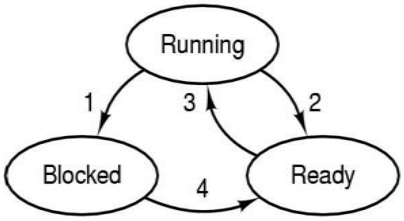
Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal



9

Process States (1)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

10

Process states

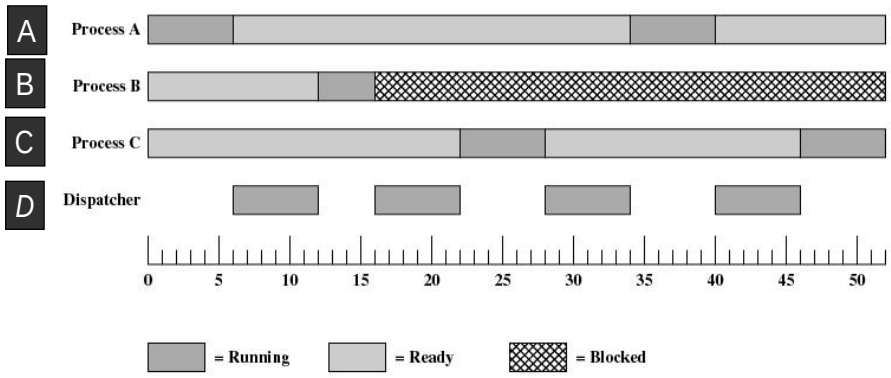
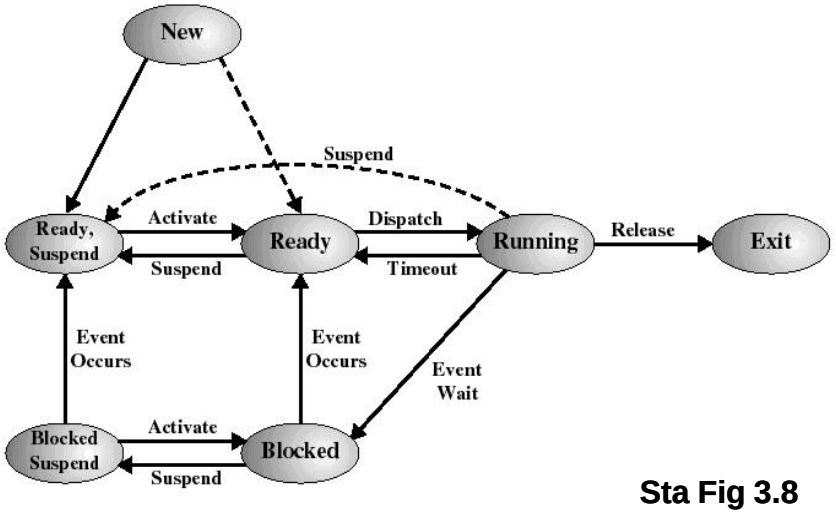


Figure 3.7 Process States for Trace of Figure 3.4

11

Process states



Sta Fig 3.8

12

Process Control Block

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Fields of a process table entry

13

Tasks on interrupt

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

The process is not always changed in step 6,
if running process can continue.

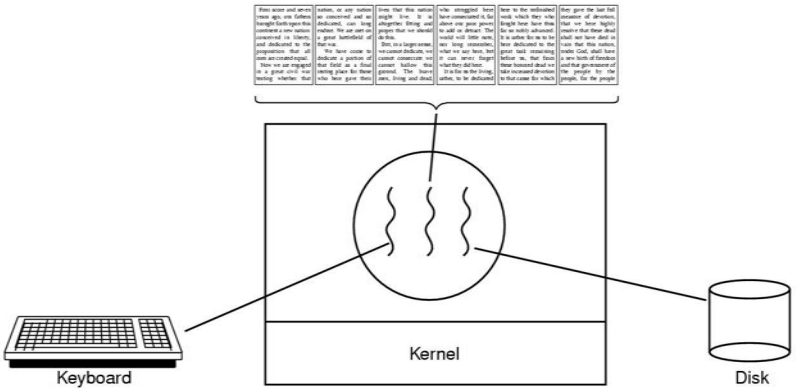
Process can be changed only at step 6.

14

Threads

15

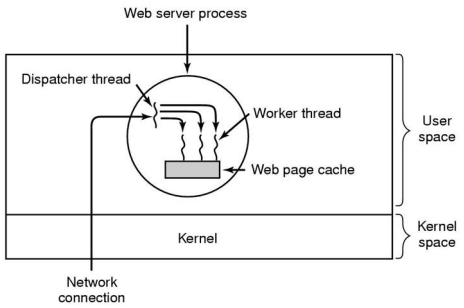
Word processor with three threads



- One thread interacts with user
- One thread handles reformatting in background
- One thread handles disk backups in background

16

Multithreaded web server using dispatcher



- Dispatcher receives requests
- Handles to an idle worker thread for processing
- Suitable model for such: finite-state machine

```
while (TRUE) {
  get_next_request(&buf);
  handoff_work(&buf);
}
```

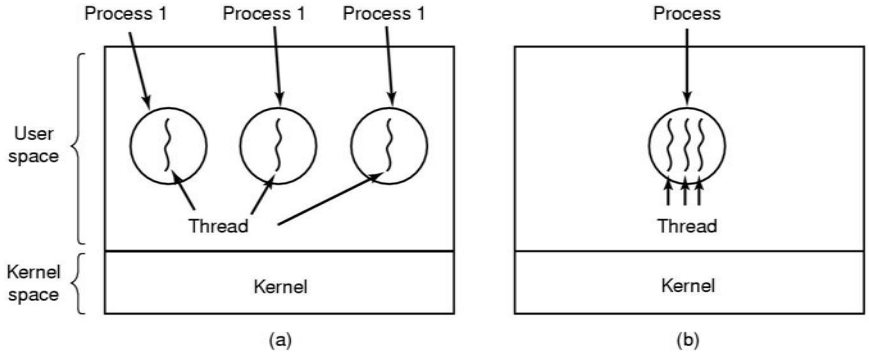
(a)

```
while (TRUE) {
  wait_for_work(&buf)
  look_for_page_in_cache(&buf, &page);
  if (page_not_in_cache(&page)
      read_page_from_disk(&buf, &page);
  return_page(&page);
}
```

(b)

17

Thread model



(a) Three processes each with one thread
(b) One process with three threads

18

Thread Model: Process vs thread

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Items shared by all threads in a process
- Items private to each thread in thread
- Each thread has its own stack!

19

Why threads?

- Multiple parallel activities. Thread provide easier programming models
- Faster to create and destroy one thread than a whole process
- When one thread waits, another might be able to execute
- Shared data area of threads in one process. Efficient resource sharing and easy communication.

BUT

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

⇨ Concurrent programming course

20

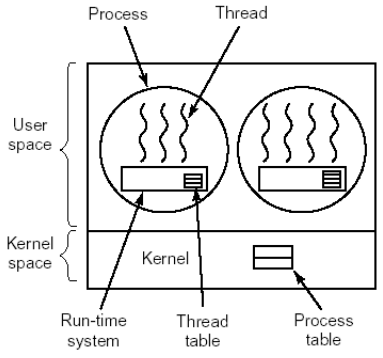
POSIX threads (pthreads)

IEEE standard for threads for portable thread programs

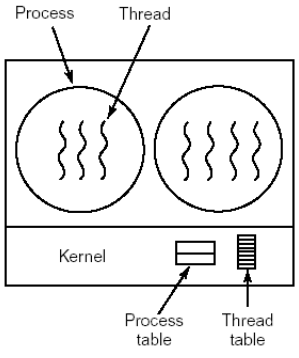
- pthread_create()
  - Create a new thread
- pthread_exit()
  - Terminate the calling thread
- pthread_join()
  - Wait for a specific thread to exit
- Pthread_yield()
  - Release the CPU to let another thread run
- Functions for synchronization and mutual exclusion
- And more than 50 other functions

21

User-level vs kernel-level threads



(a) A user-level threads package.



(b) A threads package managed by the kernel.

Tan08 Fig 2-16

- Kernel (or OS) is not aware of threads, schedules processes
- User process must dispatch threads itself

- All thread control and dispatching done by the kernel
- No control on the user level

22

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!

23

Kernel-level threads

Advantages

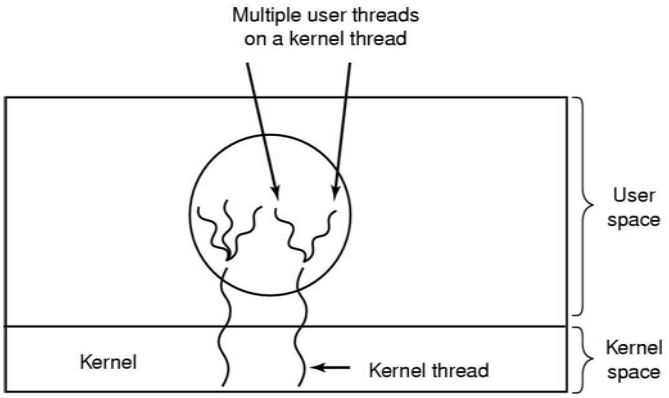
- Threads of one process can be executed simultaneously on multiple processors
- If one thread is Blocked, the other threads of this process may still continue
- Often also the kernel implementation is multithreaded

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

24

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads

25

Solaris

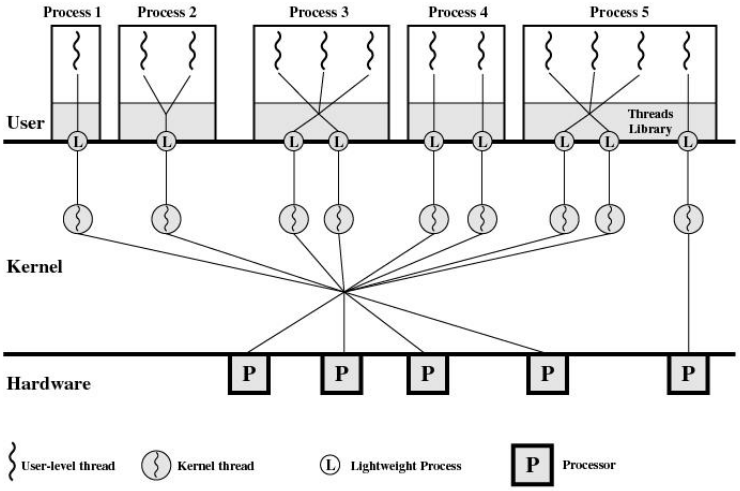


Figure 4.15 Solaris Multithreaded Architecture Example

26

Making single-threaded code multithreaded

- Global variables
  - Threads may not be aware of others using the same variable
  - See next slide for solutions
- Library procedures
  - May not be reentrant (second call to procedure before the first one is finished)
  - Solution: rewrite the library or use an excluding jacket
- Signals
  - No simple solutions – difficult already in single thread
- Stack management
  - How to increase a thread's stack in case of stack overflow

27

Threads and global variables

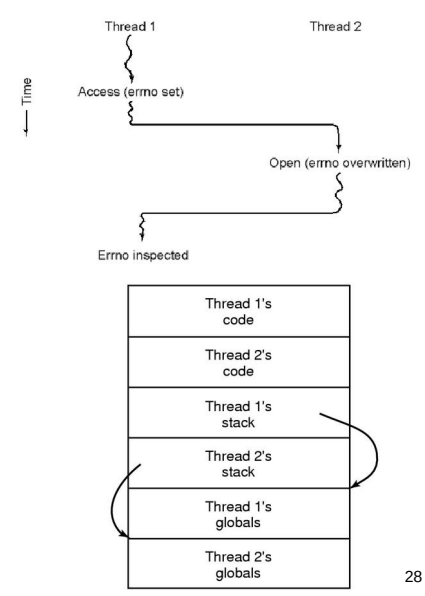
Conflicts between threads over the use of a global variable

One solution:

- Prohibit global variables

Alternative:

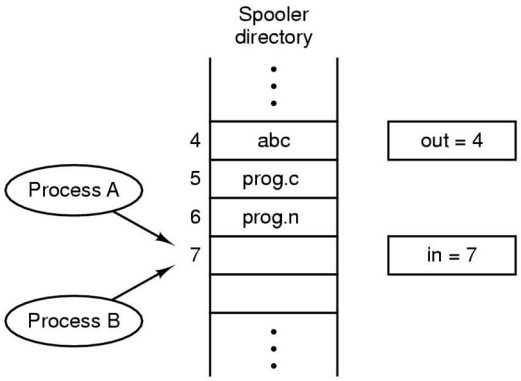
- Private global variables
- Accessing these is tricky, language may not support
- New library procedures



28

Interprocess Communication

Race Conditions



Two processes want to access shared memory at same time

29

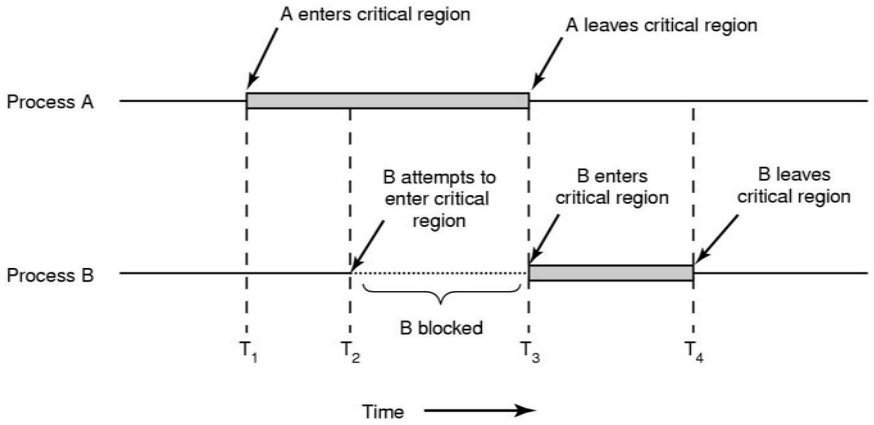
Critical Regions (1)

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
2. No assumptions made about speeds or numbers of CPUs
3. No process running outside its critical region may block another process
4. No process must wait forever to enter its critical region

30

Critical Regions (2)



Mutual exclusion using critical regions

31

Mutual Exclusion

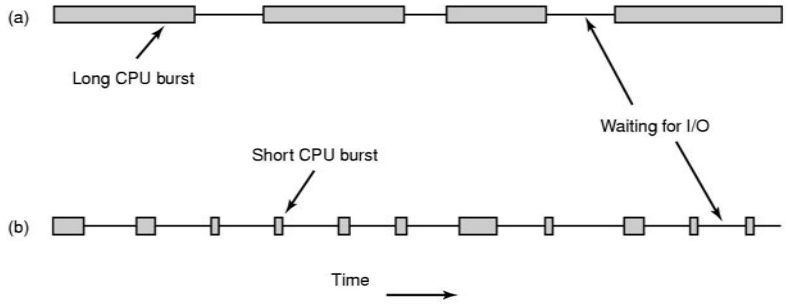
- Busy Waiting – occuby the CPU while waiting
- Sleep and Wakeup
  - Semaphores
  - Mutexes
  - Monitors
  - Message Passing
  - Barriers

Concurrent Programming
Rinnakkaisohjelmointi

32

Scheduling

Introduction to Scheduling (1)



- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

33

Introduction to Scheduling (2)

All systems

Fairness - giving each process a fair share of the CPU
Policy enforcement - seeing that stated policy is carried out
Balance - keeping all parts of the system busy

Batch systems

Throughput - maximize jobs per hour
Turnaround time - minimize time between submission and termination
CPU utilization - keep the CPU busy all the time

Interactive systems

Response time - respond to requests quickly
Proportionality - meet users' expectations

Real-time systems

Meeting deadlines - avoid losing data
Predictability - avoid quality degradation in multimedia systems

Scheduling Algorithm Goals

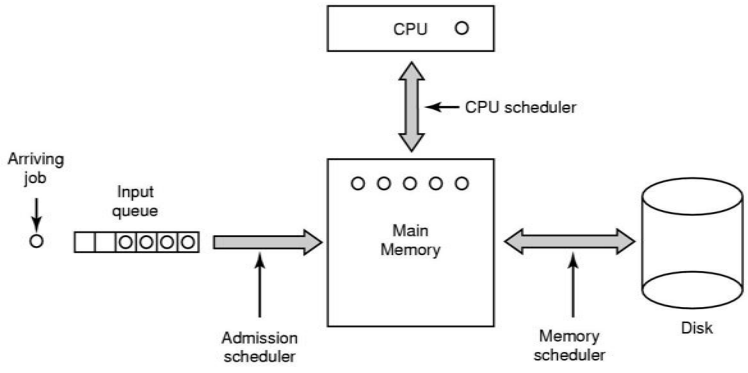
34

Scheduling in Batch Systems (1)

An example of shortest job first scheduling

35

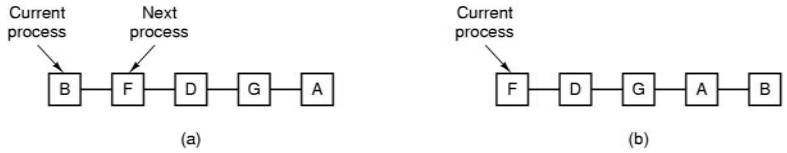
Scheduling in Batch Systems (2)



Three level scheduling

36

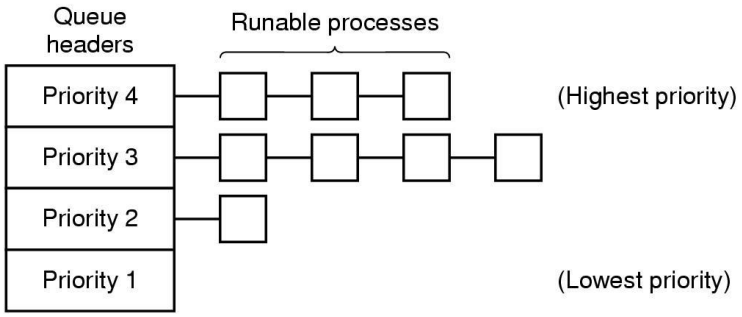
Scheduling in Interactive Systems (1)



- Round Robin Scheduling
  - list of runnable processes
  - list of runnable processes after B uses up its quantum

37

Scheduling in Interactive Systems (2)



A scheduling algorithm with four priority classes

38

Scheduling in Real-Time Systems

Schedulable real-time system

- Given
  - *m* periodic events
  - event *i* occurs within period P_i and requires C_i seconds
- Then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

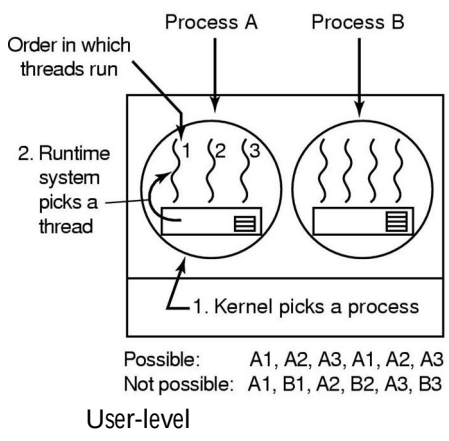
39

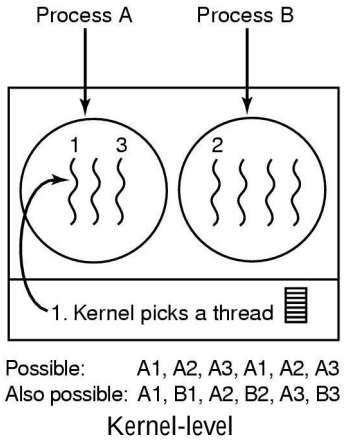
Policy versus Mechanism

- Separate what is allowed to be done with how it is done
  - a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized
  - mechanism in the kernel
- Parameters filled in by user processes
  - policy set by user process

40

Thread Scheduling (1)





Possible scheduling of threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

41