Processes and Threads
(Topic 2-1)

홍 성 수

서울대학교 공과대학 전기 공학부
Real-Time Operating Systems Laboratory

Processes and Threads

- Question: What is a process and why is it useful?
- Why?

  With many things happening at once in a system, need some way of separating them all out cleanly.
  - Important concept: Decomposition

    Solve a hard problem by chopping it into several simpler problems that can be solved separately.

Processes (1)

- What? Definition of a process:
  - An execution stream in the context of a particular process state.
- What is an “execution stream” and what is a “process state”?
- Process state is everything that can affect, or be affected by the process:
  - code, data values, open files, etc.

Processes (2)

- Execution stream is a sequence of instructions performed in a process state.
- Key simplifying feature of a process:
  - Only one thing happens at a time within a process.
- System classifications:
  - Uniprogramming: Only one process at a time.

    Mostly personal computers.

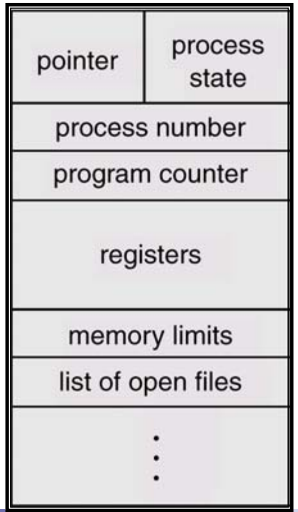
    Makes some parts of OS easier, but others hard.
  - Multiprogramming: Multiple processes at a time.

    Most systems support multiprogramming.

Processes (3)

- With multiprogramming, OS must keep track of the processes:
  - For each process, a process control block (PCB) holds:
    - Execution state (saved registers, etc.).
    - Scheduling information (priority).
    - Accounting and other misc. information (open files).
  - System-wide table of PCB: Process table.
  - Unix: Fixed-size array of PCB's

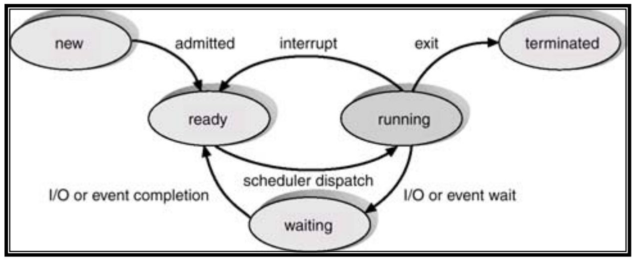
Processes (4): PCB



Processes (5): State Transition

- As a process executes, it changes state.
  - New: The process is being created.
  - Running: Instructions are being executed.
  - Waiting: The process is waiting for some event to occur.
  - Ready: The process is waiting to be assigned to CPU.
  - Terminated: The process has finished execution.

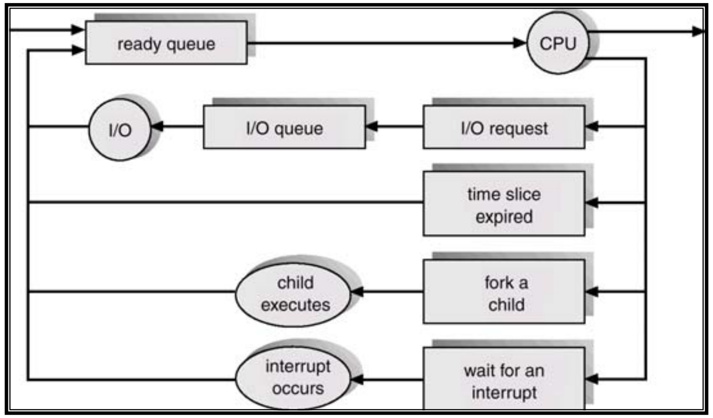
Processes (6): State Transition



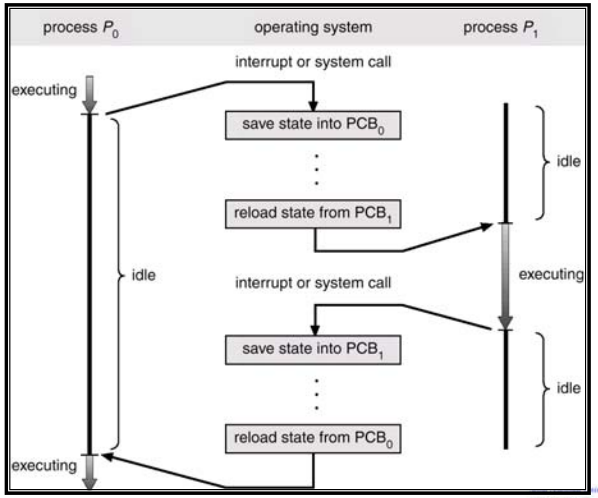
Processes (7): Scheduling Queues

- Job queue
  - Set of all processes in the system.
- Ready queue
  - Set of all processes residing in main memory, ready and waiting to execute.
- Device queues
  - Set of processes waiting for an I/O device.
- Process migration between the various queues.

Processes (8): Scheduling



Processes (9): Scheduling



Process Scheduling (1)

- For several processes to share a CPU, the OS must have:
  - Fair scheduling

    Make sure each process gets a chance to run.
  - Protection

    Making sure processes don't trash each other.

Process Scheduling (2)

- Dispatcher: Inner-most portion of the OS that runs processes:

  loop forever {

  1) Run the process for a while.

  2) Process and save its state.

  3) Load state of another process.

  }

Process Scheduling (3)

- Dispatcher policies and mechanisms:
  - How does the dispatcher keep control?

    CPU can only be doing one thing at a time.

    User process running means that dispatcher isn't.
  - Which process is executed next?

    Need to locate runnable processes efficiently.

Process Scheduling (4)

- How does the dispatcher regain control?

  (1)Trust the process to wake the dispatcher up.

  Problem: Sometimes processes misbehave.

  (2) Provide the dispatcher with an alarm clock.

  Timer hardware and interrupts.

Process Scheduling (5)

- Control returns to OS on:
  - Traps: Events internal to the user processes:

    System calls.

    Errors (illegal instructions, address error, etc).

    Page faults.
  - Interrupts: Events external fro the user process:

    Character typed at a terminal.

    Completion of a disk transfer.

    Timer: to make sure OS eventually gets control.

Process Scheduling (6)

- Once dispatcher gets control how to decide who's next?

  Possibilities:

  (1) Scan process table for first runnable process:

  – Might spend much time searching.

  – Results in weird priorities: Small PIDs better.

    Question: How do you know a process is runnable?

Process Scheduling (7)

(2) Link together the runnable processes into a queue.

– Dispatcher takes from the head of the queue.

– Runnable processes are inserted at back of queue.

  Called "Ready list" or "Run queue."

(3) Assign priorities to processes.

– Keep queue sorted by priority.

– Separate queue per priority.

Process Scheduling (8)

- Who decides priorities and how are priorities chosen?
  - Who? A separate part of the OS: the scheduler

    Question: Why not by the dispatcher?

    Concept: Separation of mechanism and policy.
  - How? Subject of the next topic (Topic 3).

Context Switching (1)

- How does the dispatcher save and restore state?

  Mechanism — Context switch
- What must get saved?
  - Everything that next process could or will damage:

    Program counter,

    Processor status word (condition codes, etc.),

    General purpose registers, floating-point registers.

    All of memory?

Context Switching (2)

- Memory could be large so saving it could be expensive

  Possibilities:

  (1) Don't save memory — Trust next process.

  - Multiprogramming on PC and Mac.
  - Called threads or lightweight-processes.

Context Switching (3)

(2) Save all memory to disk.

- Also, an early personal computer/ workstation.
- Assume disk transfers at one megabyte/ second.

  How long does saving a 4 megabyte process take?

(3) Protection memory from next process:

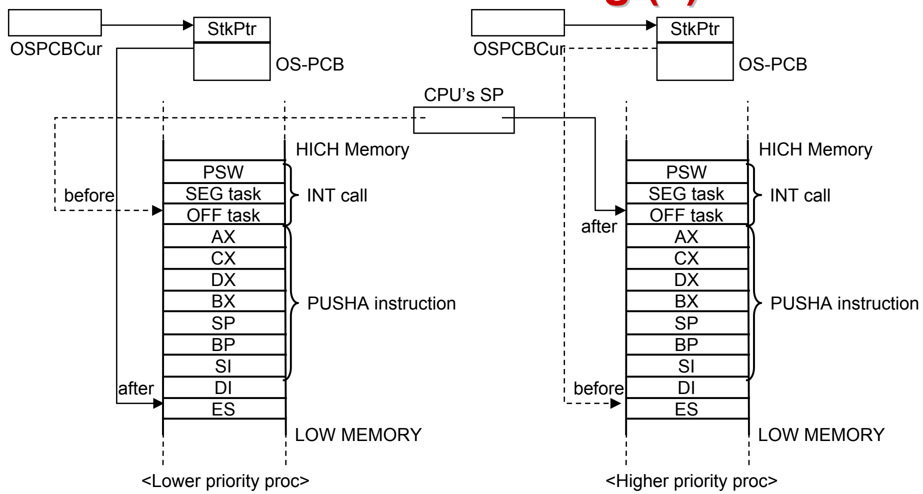
- Memory management — Topic 6.

Context Switching (4)

- Context switching implementation:
  - Machine dependent: Different for MIPS, SPARC, 386, etc.
  - Tricky:
    - OS must execute code to save state without changing the process' state.
    - Requires some special hardware support.

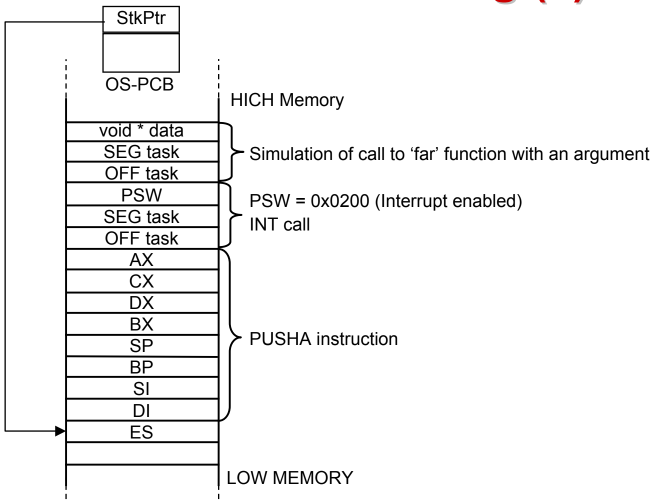
      Example: Save PC and PSR on trap or interrupt.

Context Switching (5)



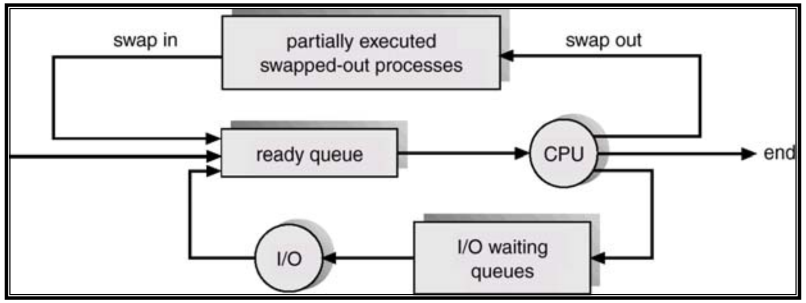
Stack frames during a context Switch (80186/80188 Small)

Context Switching (6)



Stack frame when procs are created (80186/80188 Small)

Short vs Long Term Scheduler



Process Creation (1)

- Creating new processes:
  - Build one from scratch; (e.g. Unix Process 0).
  - Clone an existing one; (e.g. Unix fork() syscall).

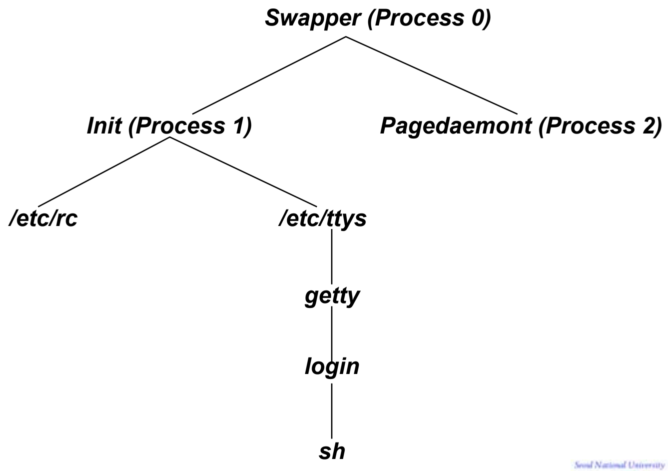
Process Creation (2)

- From scratch:
  1) Load code and data into memory.
  2) Create (empty) call stack.
  3) Create and initialize a process control block.
  4) Put process on ready list.

Process Creation (3)

- Cloning: Unix fork() system call.
  1) Stop current process and save its state.
  2) Make a copy of code, data, stack, and PCB.
  3) Add new PCB to ready list.
  Not quite right. What's missing?
  — Return to the child and the parent.
- Process creation in Unix with fork() and exec():

Process Creation in Unix (4)



Process Creation (5)

- Shell example:

```
for(;;) {
  cmd = readcmd();
  pid = fork();
  if(pid == 0) {
    // Child – Setup environment.
    exec(cmd);
    exit(1);
  } else {
    //Parent – Wait for command to finish.
    wait(pid);
  }
}
```


Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**).
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
    - Operating system does not allow child to continue if its parent terminates.
    - Cascading termination.

Process Characteristics (1)

- Unit of resource ownership - process is allocated:
  - a virtual address space to hold the process image
  - control of some resources (files, I/O devices...)
- Unit of dispatching - process is an execution path through one or more programs
  - execution may be interleaved with other process
  - the process has an execution state and a dispatching priority

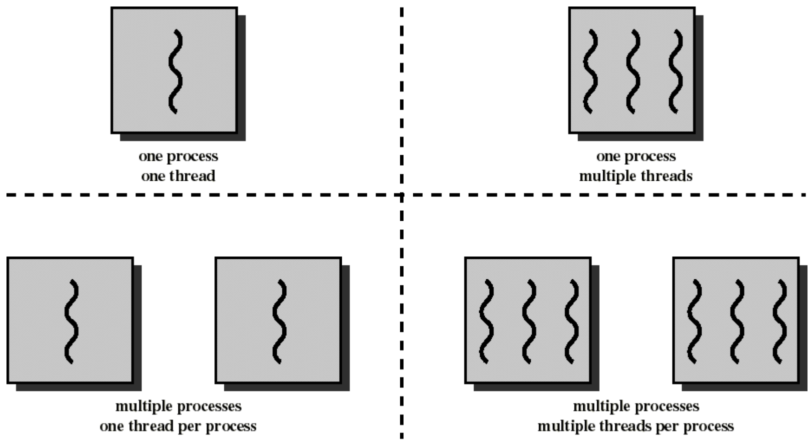
Process Characteristics (2)

- These two characteristics are treated independently by some recent OS
- The unit of dispatching is usually referred to a thread or a lightweight process
- The unit of resource ownership is usually referred to as a process or task

Multithreading vs. Single threading

- Multithreading: when the OS supports multiple threads of execution within a single process
- Single threading: when the OS does not recognize the concept of thread
- MS-DOS support a single user process and a single thread
- UNIX supports multiple user processes but only supports one thread per process
- Solaris supports multiple threads

Threads and Processes



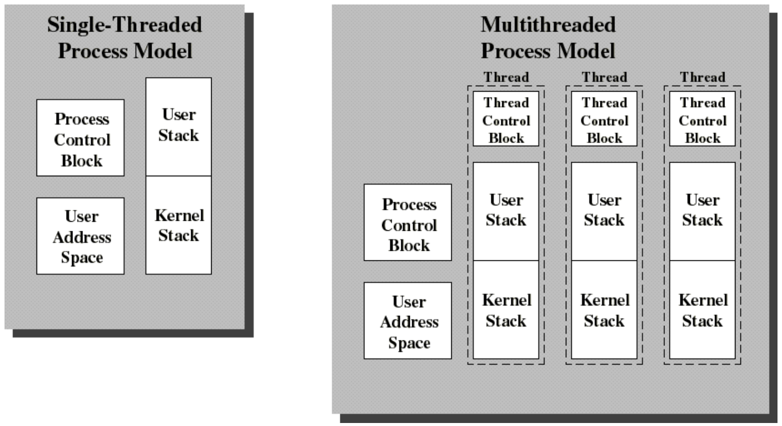
Processes

- Have a virtual address space which holds the process image
- Protected access to processors, other processes, files, and I/O resources

Threads

- Has an execution state (running, ready, etc.)
- Saves thread context when not running
- Has an execution stack and some per-thread static storage for local variables
- Has access to the memory address space and resources of its process
  - all threads of a process share this
  - when one thread alters a (non-private) memory item, all other threads (of the process) sees that
  - a file open with one thread, is available to others

Single Threaded and Multithreaded Process Models



Thread Control Block contains a register image, thread priority and thread state information

Benefits of Threads vs Processes

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process

Benefits of Threads

- Example 1: a file server on a LAN
- It needs to handle several file requests over a short period
- Hence more efficient to create (and destroy) a single thread for each request
- On a SMP machine: multiple threads can possibly be executing simultaneously on different processors
- Example 2: one thread displays menu and reads user input while the other thread execute user commands

Application Benefits of Threads

- Consider an application that consists of several independent parts that do not need to run in sequence
- Each part can be implemented as a thread
- Whenever one thread is blocked waiting for an I/O, execution could possibly switch to another thread of the same application (instead of switching to another process)

Benefits of Threads

- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel
- Therefore necessary to synchronize the activities of various threads so that they do not obtain inconsistent views of the data

Example of Inconsistent View

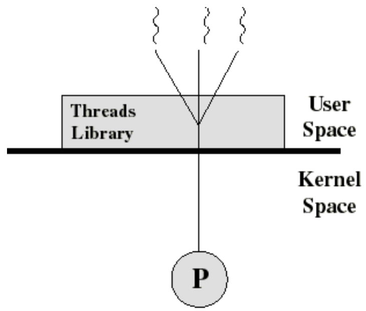
- 3 variables: A, B, C which are shared by thread T1 and thread T2
- T1 computes C = A+B
- T2 transfers amount X from A to B
  - T2 must do: A = A -X and B = B+X (so that A+B is unchanged)
- But if T1 computes A+B after T2 has done A = A-X but before B = B+X
- then T1 will not obtain the correct result for C = A + B

Threads States

- Three key states: running, ready, blocked
- They have no suspend state because all threads within the same process share the same address space
  - Indeed: suspending (ie: swapping) a single thread involves suspending all threads of the same process
- Termination of a process, terminates all threads within the process

User-Level Threads (ULT)

- The kernel is not aware of the existence of threads
- All thread management is done by the application by using a thread library
- Thread switching does not require kernel mode privileges (no mode switch)
- Scheduling is application specific



Threads Library

- Contains code for:
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts

Kernel Activity for ULTs

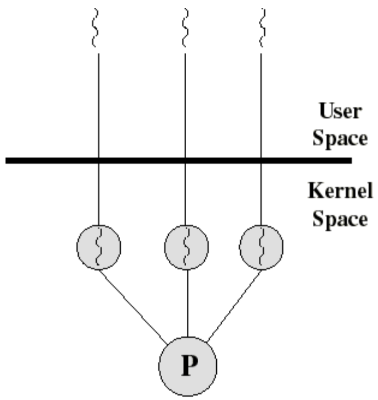
- The kernel is not aware of thread activity but it is still managing process activity
- When a thread makes a system call, the whole process will be blocked
- but for the thread library that thread is still in the running state
- So thread states are independent of process states

Advantages and Inconveniences of ULT

- Advantages
  - ❑ Thread switching does not involve the kernel: no mode switching
  - ❑ Scheduling can be application specific: choose the best algorithm.
  - ❑ ULTs can run on any OS. Only needs a thread library
- Inconveniences
  - ❑ Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked
  - ❑ The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors

Kernel-Level Threads (KLT)

- All thread management is done by kernel
- No thread library but an API to the kernel thread facility
- Kernel maintains context information for the process and the threads
- Switching between threads requires the kernel
- Scheduling on a thread basis
- Ex: Windows NT and OS/2

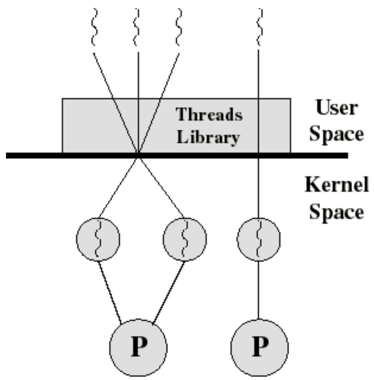


Advantages and inconveniences of KLT

- Advantages
  - ❑ the kernel can simultaneously schedule many threads of the same process on many processors
  - ❑ blocking is done on a thread level
  - ❑ kernel routines can be multithreaded
- Inconveniences
  - ❑ thread switching within the same process involves the kernel. We have 2 mode switches per thread switch
  - ❑ this results in a significant slow down

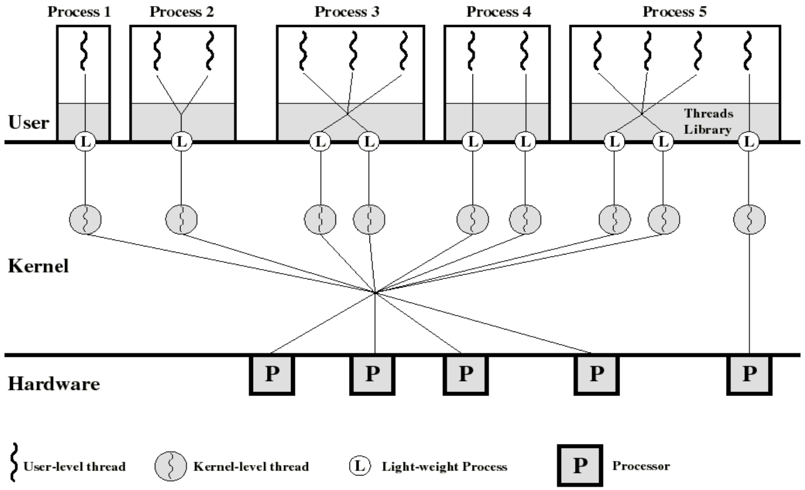
Combined ULT/KLT Approaches

- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- May combine the best of both approaches
- Example is Solaris



Solaris

- Process includes the user's address space, stack, and process control block
- User-level threads (threads library)
  - invisible to the OS
  - are the interface for application parallelism
- Kernel threads
  - the unit that can be dispatched on a processor and it's structures are maintain by the kernel
- Lightweight processes (LWP)
  - each LWP supports one or more ULTs and maps to exactly one KLT
  - each LWP is visible to the application



Process 2 is equivalent to a pure ULT approach
Process 4 is equivalent to a pure KLT approach
We can specify a different degree of parallelism (process 3 and 5)

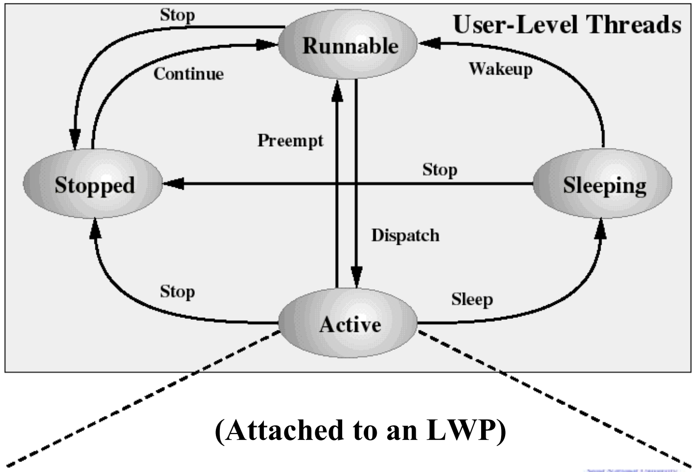
Solaris: Versatility

- We can use ULTs when logical parallelism does not need to be supported by hardware parallelism (we save mode switching)
  - Ex: Multiple windows but only one is active at any one time
- If threads may block then we can specify two or more LWPs to avoid blocking the whole application

Solaris: User-Level Thread Execution

- Transitions among these states is under the exclusive control of the application
  - a transition can occur only when a call is made to a function of the thread library
- It's only when a ULT is in the active state that it is attached to a LWP (so that it will run when the kernel level thread runs)
  - a thread may transfer to the sleeping state by invoking a synchronization primitive and later transfer to the runnable state when the event waited for occurs
  - A thread may force another thread to go to the stop state...

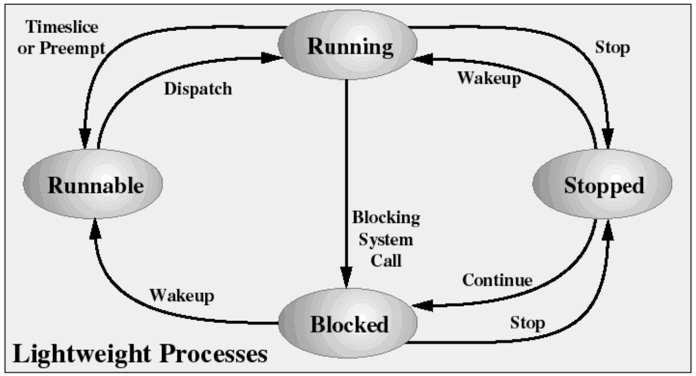
Solaris: User-Level Thread States



Decomposition of User-Level Active State

- When a ULT is active, it is associated to an LWP and, thus, to a KLT
- Transitions among the LWP states is under the exclusive control of the kernel
- An LWP can be in the following states:
  - running: when the KLT is executing
  - blocked: because the KLT issued a blocking system call (but the ULT remains bound to that LWP and remains active)
  - runnable: waiting to be dispatched to CPU

Solaris: Lightweight Process States



LWP states are independent of ULT states (except for bound ULTs)