CSE 306/506 Operating Systems
Threads

YoungMin Kwon

Processes and Threads

- Two characteristics of a process
  - Resource ownership
    - Virtual address space (program, data, stack, PCB…)
    - Main memory, I/O devices, files
  - Scheduling/execution
    - Execution of a process follows an execution path
    - Execution may be interleaved with that of other processes

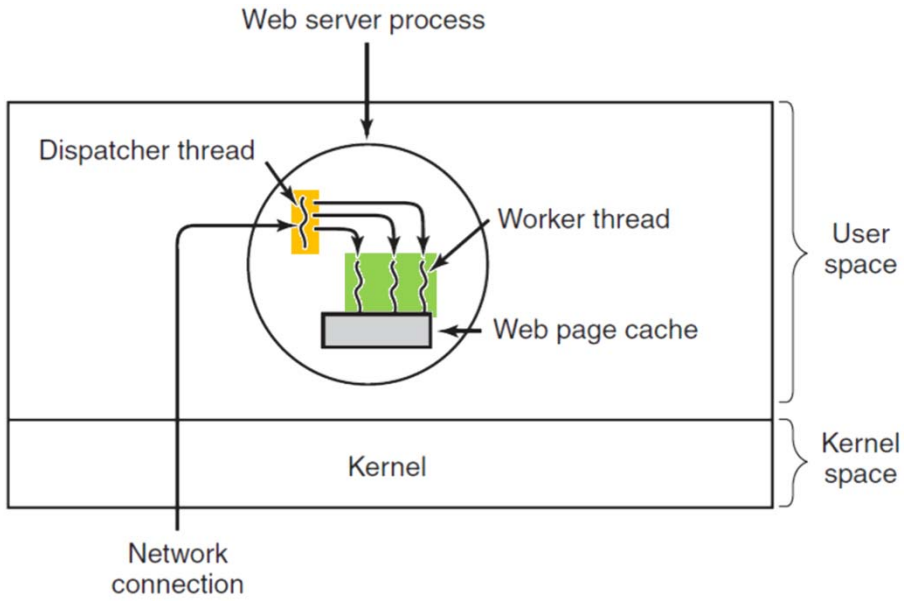
Processes and Threads

- Resource ownership and Scheduling/execution
  - Could be treated independently by the OS
- Thread (lightweight process): the unit of dispatching
- Process (task): the unit of resource ownership

Why Threads

- Parallel execution
  - Without relying on interrupts, timers, context switches
  - Parallel entities sharing an address space and data
- Easier and faster to create and destroy than processes
  - 10 ~ 100 times faster
- Performance gain
  - Not much for CPU bounded applications
  - Substantial for I/O bounded applications
- Real parallelism with multiple CPUs

A Multithreaded Web Server




```
Dispatcher thread

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

```
Worker thread

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);
}
```

Multithreading

- Multithreading
  - The ability of an OS to support multiple concurrent paths of execution within a single process
- Processes in a multithreaded environment
  - Unit of resource allocation
  - Unit of protection

Multithreading

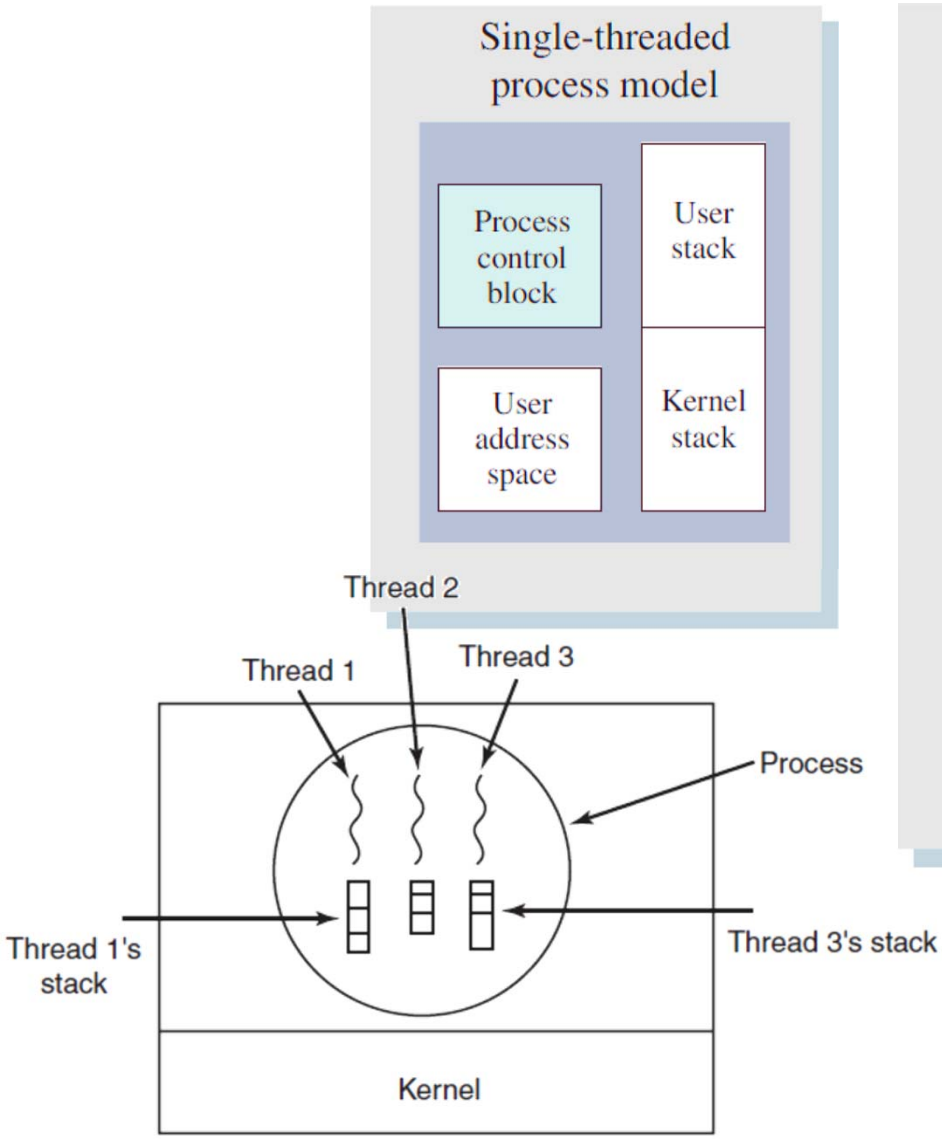
- Threads within a process have
  - Thread execution state (Running, Ready, …)
  - Saved thread context (PC, registers, …)
  - Execution stack
  - Per-thread static storage for local variables
  - Access to resources shared with other threads in the process

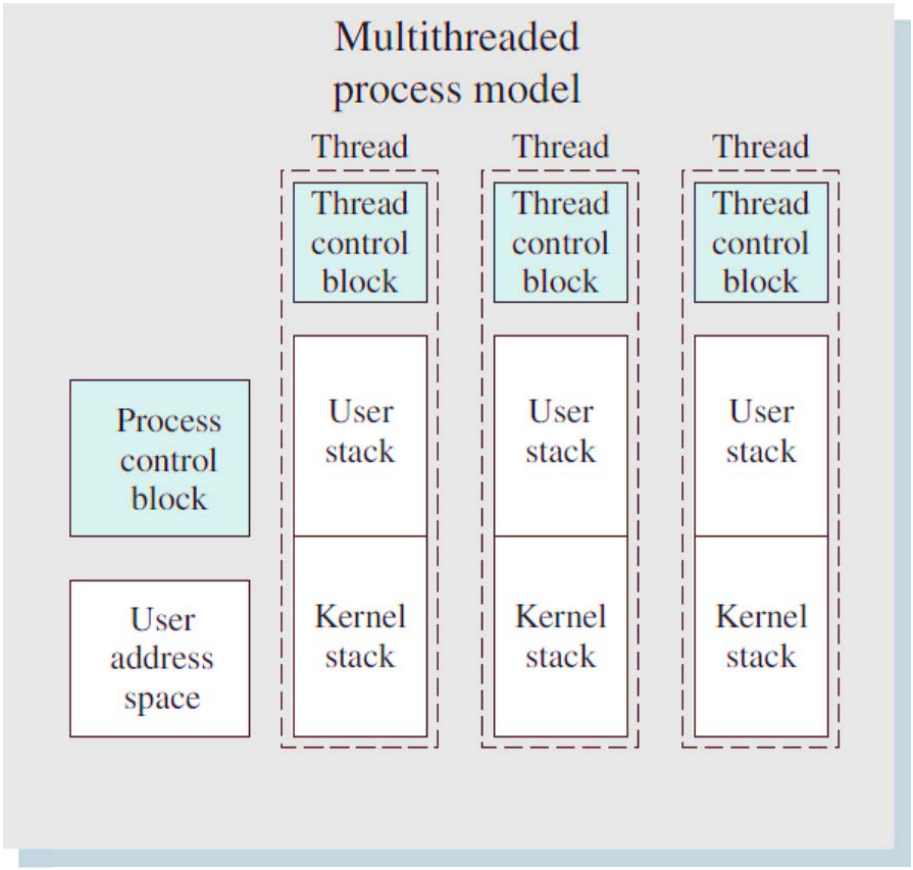
Multithreading

- All threads of a process share the states and resources of the process
  - If a thread alters data, other threads will see the change
  - If a thread opens a file with a read privilege, others can also read the file

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	

Multithreading





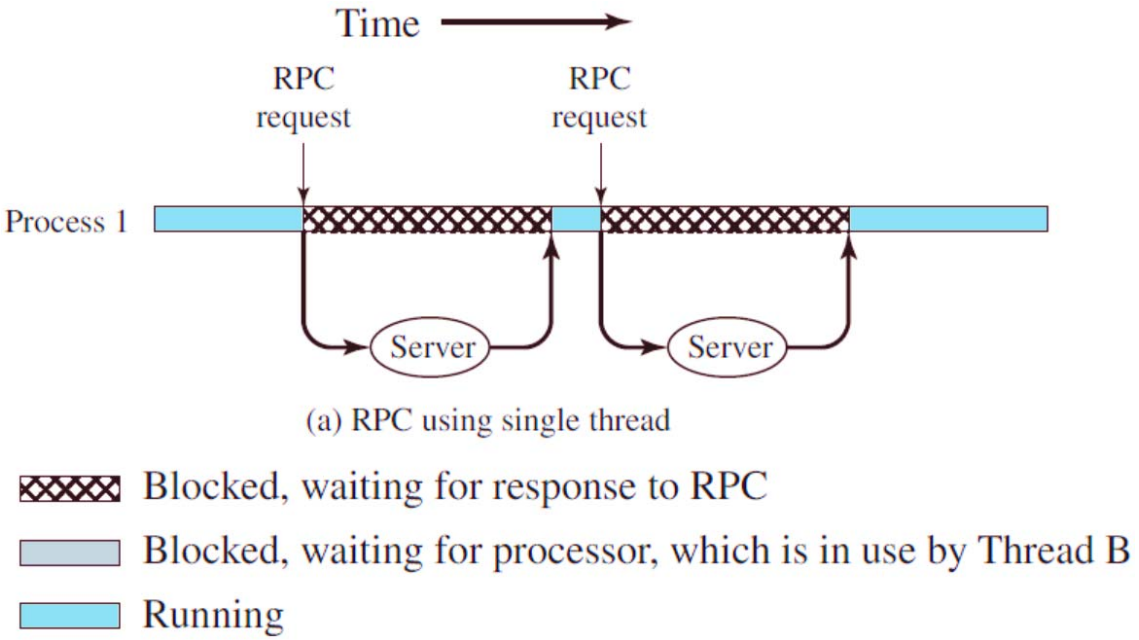
Multithreading

- Performance benefits
  - It takes far less time to create a new thread than to create a new process
  - Terminating a thread is faster than terminating a process
  - Switch between threads is faster than switch between processes
  - Threads enhance efficiency in communication between executing programs
    - No need for the protection and communication mechanisms

Thread Functionality

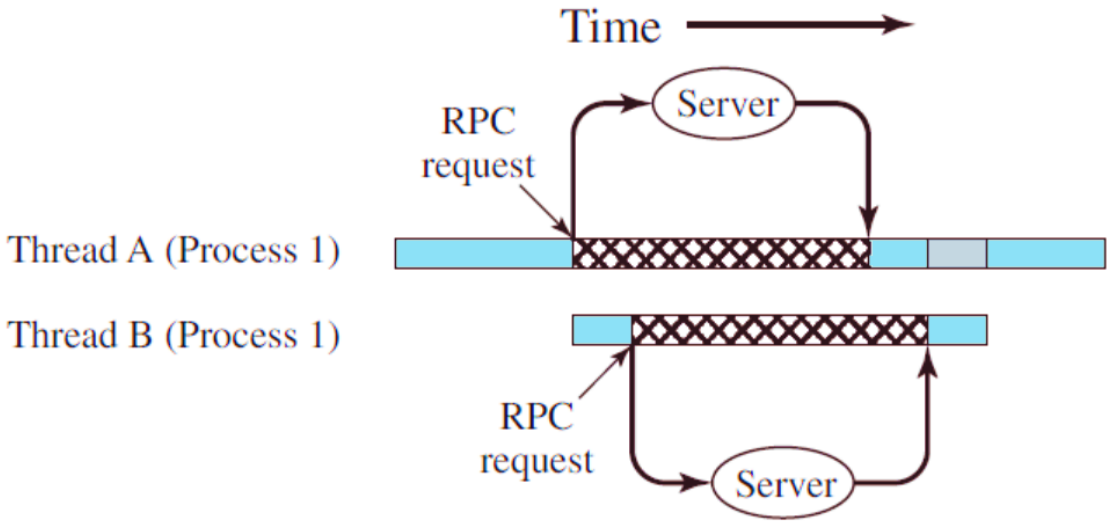
- Thread states: Running, Ready, Blocked
  - Suspend is for processes: if a process is swapped out, all of its threads are swapped out

- RPC (Remote Procedure Call) example



Thread Functionality

- RPC example



(b) RPC using one thread per server (on a uniprocessor)

Blocked, waiting for response to RPC

Blocked, waiting for processor, which is in use by Thread B

Running

Thread Functionality

- Four basic thread operations
  - Spawn: when a new process is created or when a thread spawns a new thread
    - New thread gets its own register context, stack and placed on the Ready queue
  - Block: waits for an event
  - Unblock: when the event occurs, the thread is moved to the Ready queue
  - Finish: completes its operation

POSIX Threads (Pthreads)

- A standard interface for manipulating threads from C programs
- Defines about 60 functions that allow programs
  - to create, kill, and reap threads
  - to share data safely with peer threads
  - to notify peers about changes in the system state

Thread call	Description
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

Pthreads: Creating Threads

```
#include <pthread.h>
typedef void *(func)(void *);

// To create a new thread
int pthread_create(pthread_t *tid,
                   pthread_attr_t *attr,
                   func *f,
                   void *arg);

// To get the thread id of its own
pthread_t pthread_self(void);
```

Pthreads: Terminating Threads

```
#include <pthread.h>

void pthread_exit(void *thread_return);
// Terminate explicitly
// If main thread calls pthread_exit,
//    it will wait for all other peer threads to
//    terminate, terminate itself, and terminate
//    the process

int pthread_cancel(pthread_t tid);
// Terminate the thread with the ID tid
```

Pthreads: Reaping Terminated Threads

```
#include <pthread.h>
int pthread_join(pthread_t tid,
                 void **thread_return);

// - blocks until thread tid terminates,
// - update thread_return to point to the return
//   value of the thread routine,
// - reap the memory resource
```

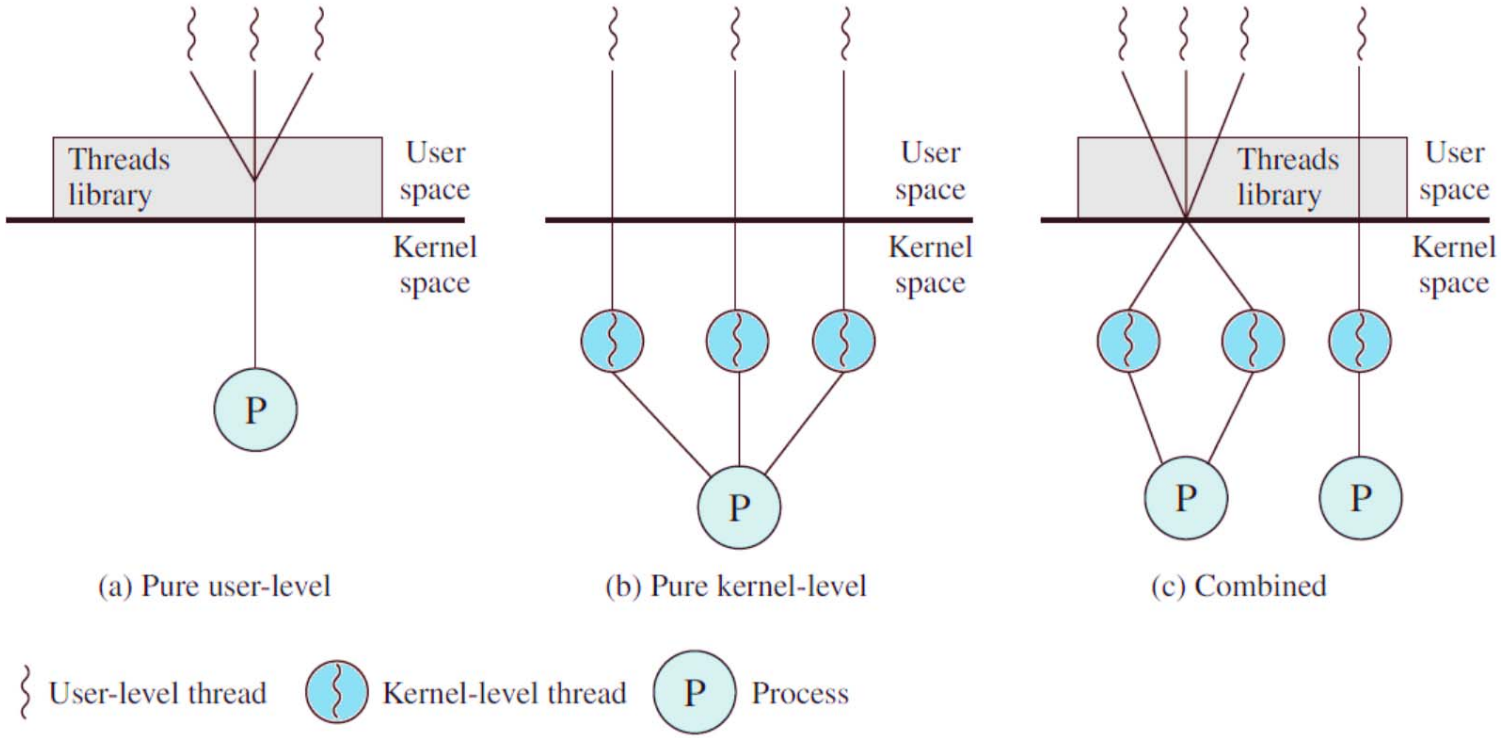
```c
#include <pthread.h>
#include <stdio.h>

void *thread(void *vargp) {
    printf("%s\n", (char*)vargp);
    pthread_exit("world");
    //return "world";
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, "hello");

    void *ret;
    pthread_join(tid, &ret);
    printf("%s\n", (char*)ret);
    return 0;
}
```

Thread Synchronization

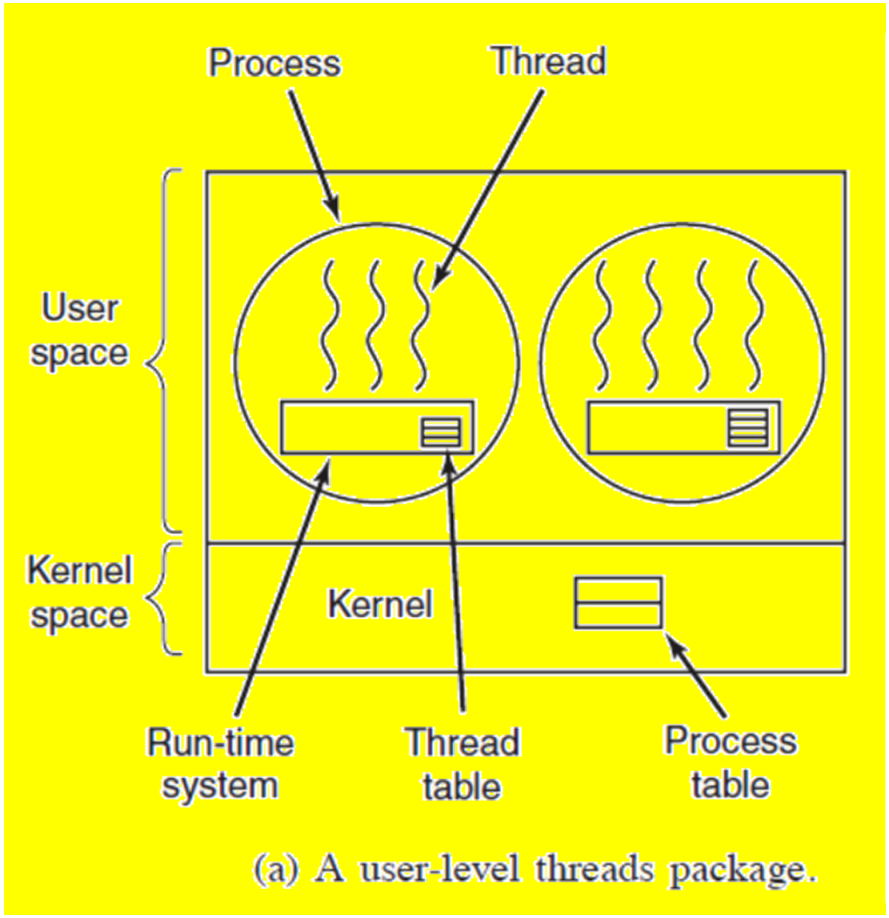
- All threads of a process share the same address space and resources
  - Any alteration of a resource by one thread affects the environment of the other threads
  - Need to synchronize the activities of various threads

Types of Threads

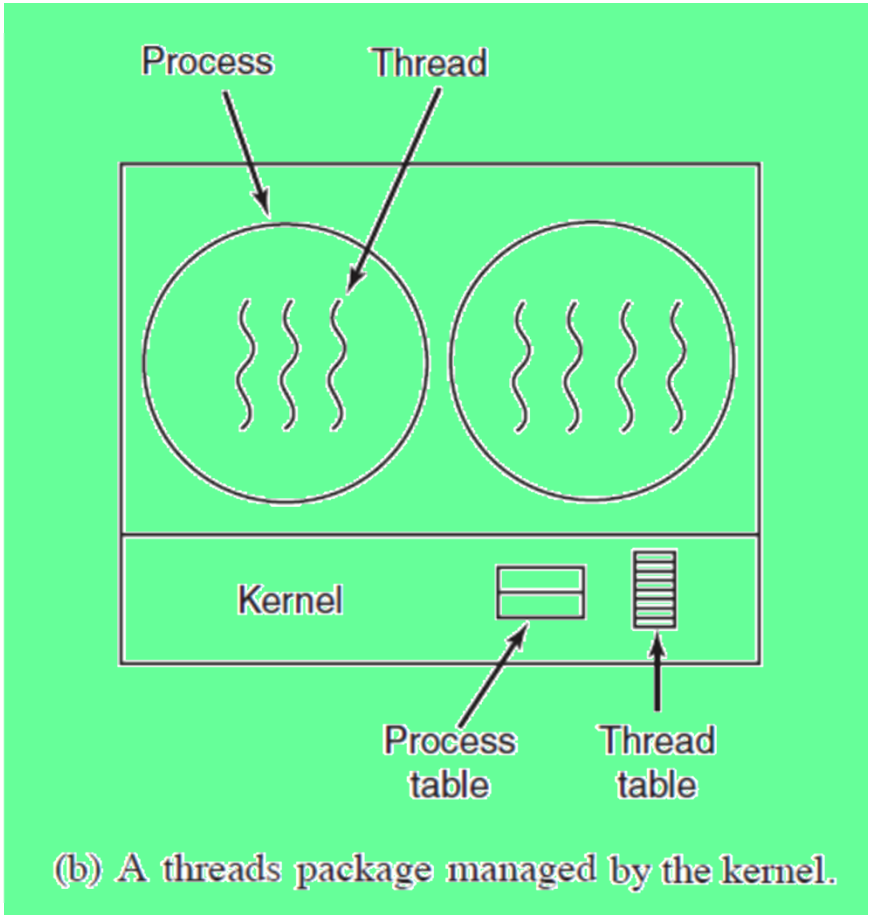
- Categories of thread implementation
  - User-level threads (ULTs)
  - Kernel-level threads (KLTs)



Types of Threads



(a) A user-level threads package.

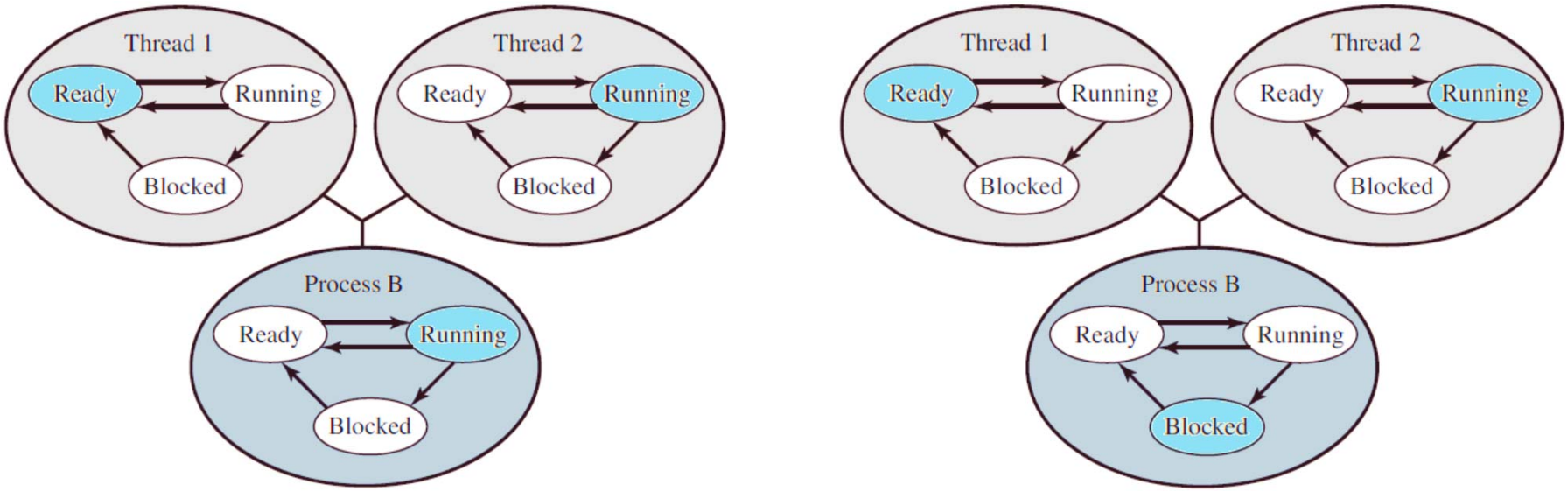


(b) A threads package managed by the kernel.

User-Level Threads

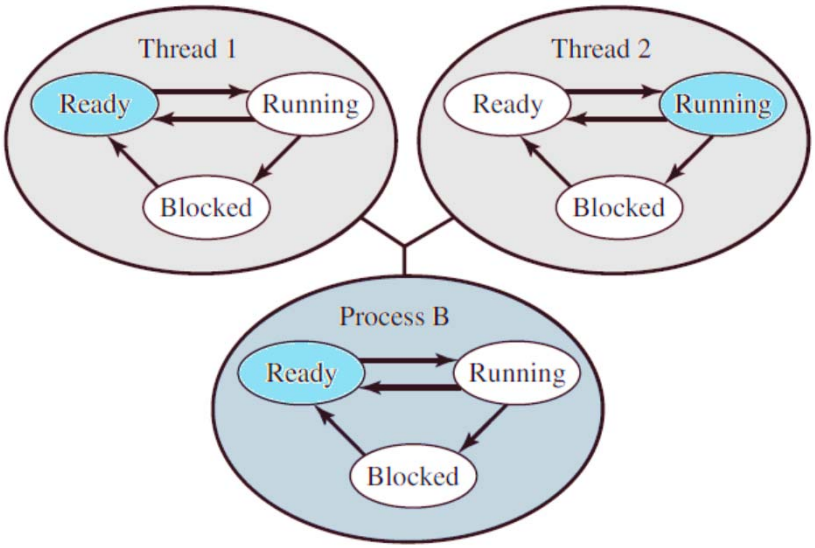
- Pure ULTs
  - Thread management is done by the application
  - Kernel is not aware of the existence of threads
  - Spawning:
    - Create a data structure for the thread
    - Pass control to a thread in the Ready state
  - Context switch:
    - Context comprises user registers, PC, stack pointer
    - When control is passed to the library: save the context
    - When control is passed from the library: restore the context

ULT: system call from a thread



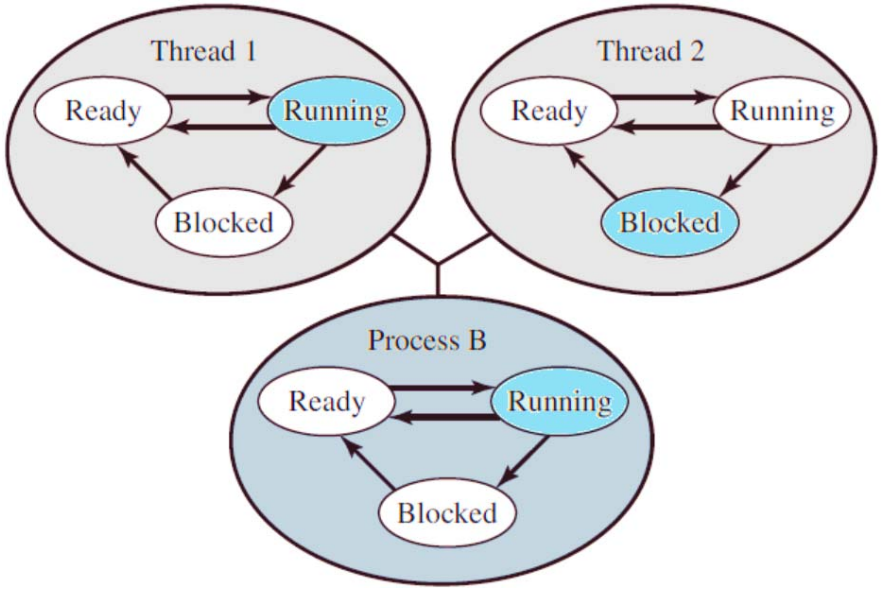
- Thread 2 makes a blocking system call
- It blocks process B
  - Thread Lib. thinks that thread 2 is in Running state

ULT: process' time slice is expired



- Process B's time slice is expired and B is moved to the Ready queue
- Process B is in the Ready state
- Thread Lib. thinks that thread 2 is in Running state

ULT: blocked thread



- Thread 2 waits for an event from thread 1
  - Thread 2 transitions to the Blocked state
  - Thread 1 transitions to the Running state

ULT vs KLT

- Advantages of ULT compared to KLT
  - Thread switching does not require kernel-mode privilege
    - Saves 2 mode switches: user → kernel, kernel → user
  - Scheduling can be application specific
  - Portability: ULTs can run on any OS

Thread and Process Operation Latencies (μs)

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

ULT vs KLT

- Disadvantages of ULT compared to KLT
  - Blocking system calls block not only the thread but all other threads in the process as well
  - Cannot take advantage of multiprocessors
- Jacketing
  - Converts blocking system calls into non-blocking ones
  - Use jacket routines that check if the device is busy
    - If it is, make the thread blocked and schedule another thread
    - Check the device again later

Kernel-Level Threads

- Thread management is done by the kernel
  - Process context and thread context are managed by the kernel
- Scheduling is done by the kernel on a thread basis
  - Kernel can schedule multiple threads of a process to multiple processors
  - If a thread is blocked, kernel can schedule another thread of the same process

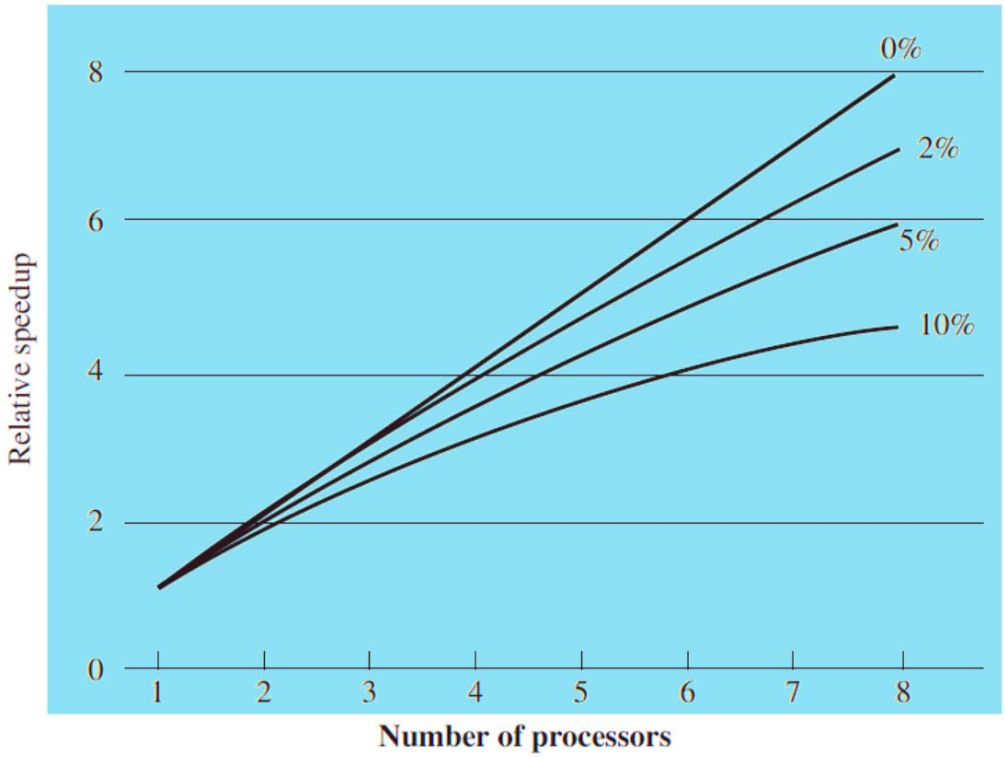
Performance Improvements on Multicore Systems

- Amdahl's law

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1-f) + \frac{f}{N}}$$

  - *1 - f*: fraction of execution time that is inherently serial

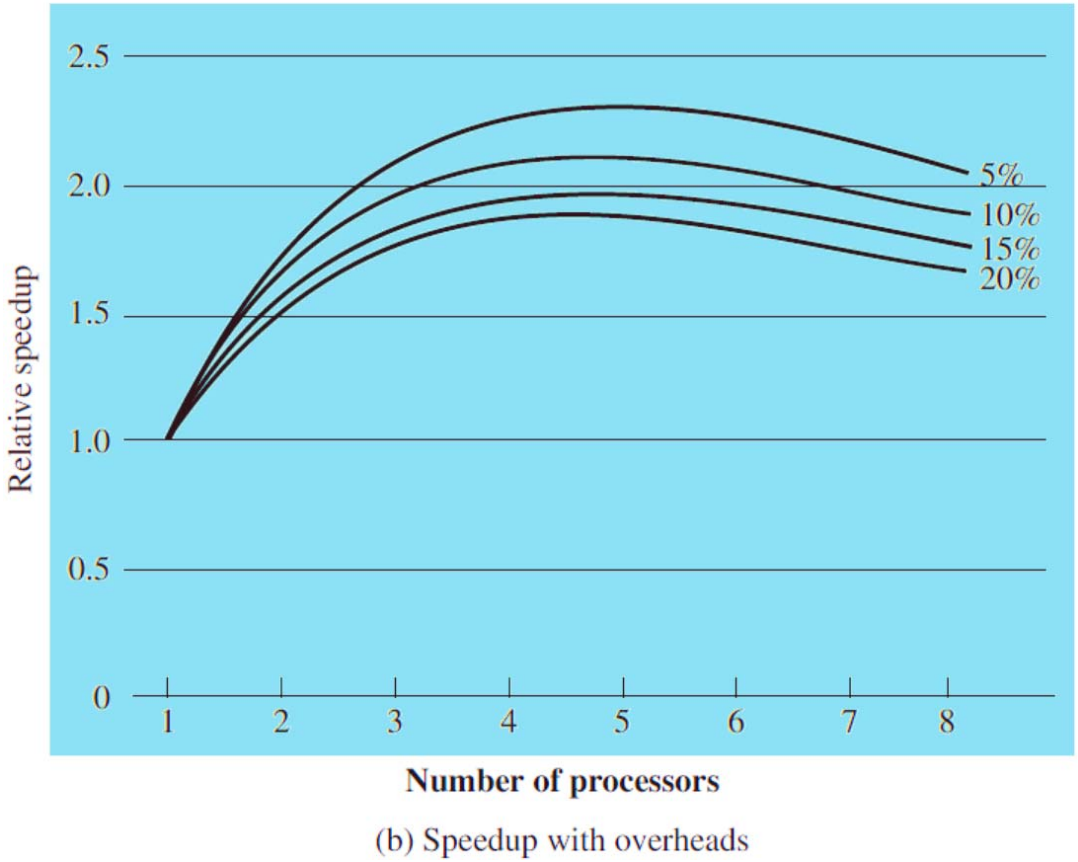
Performance Improvements on Multicore Systems



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

- Small amount of serial code has a noticeable impact

Performance Improvements on Multicore Systems



(b) Speedup with overheads

- The curves peak when considering the overhead
  - Communication, Distribution, Cache coherence, …