Review 02

Yajin Zhou (http://yajin.org)

Zhejiang University

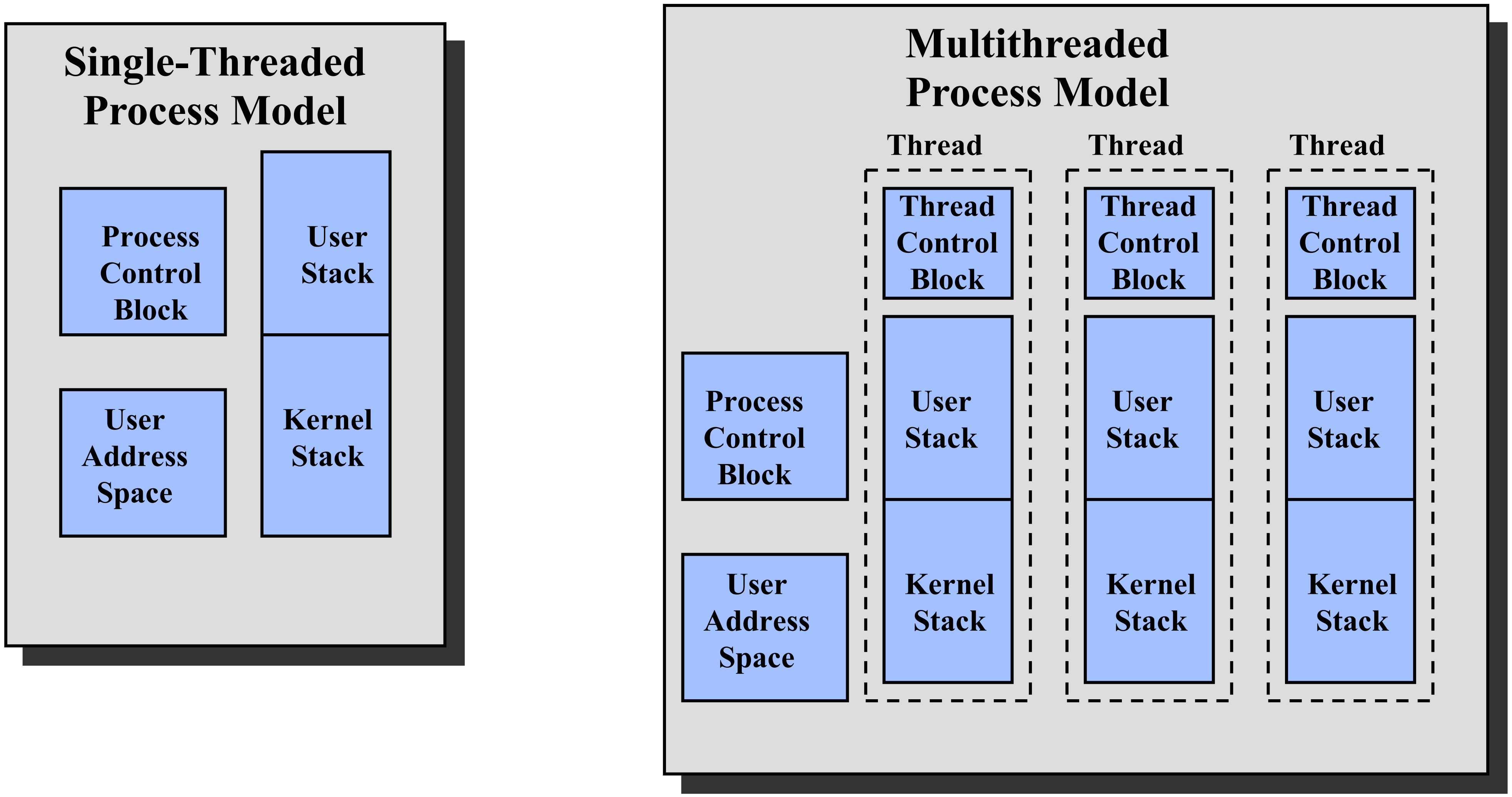
Acknowledgement: Some slides are based on Zhi Wang (FSU)'s slides.

04: Thread

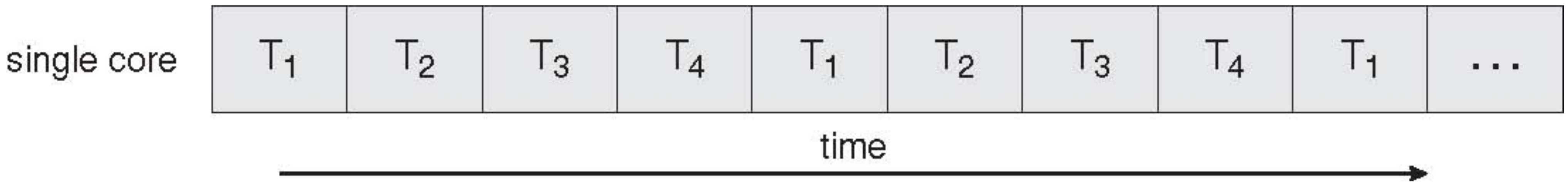
Motivation

- Why threads?
  - multiple tasks of an application can be implemented by threads
    - e.g., update display, fetch data, spell checking, answer a network request
  - process creation is heavy-weight while thread creation is light-weight - why?
  - threads can simplify code, increase efficiency
- Kernels are generally multithreaded

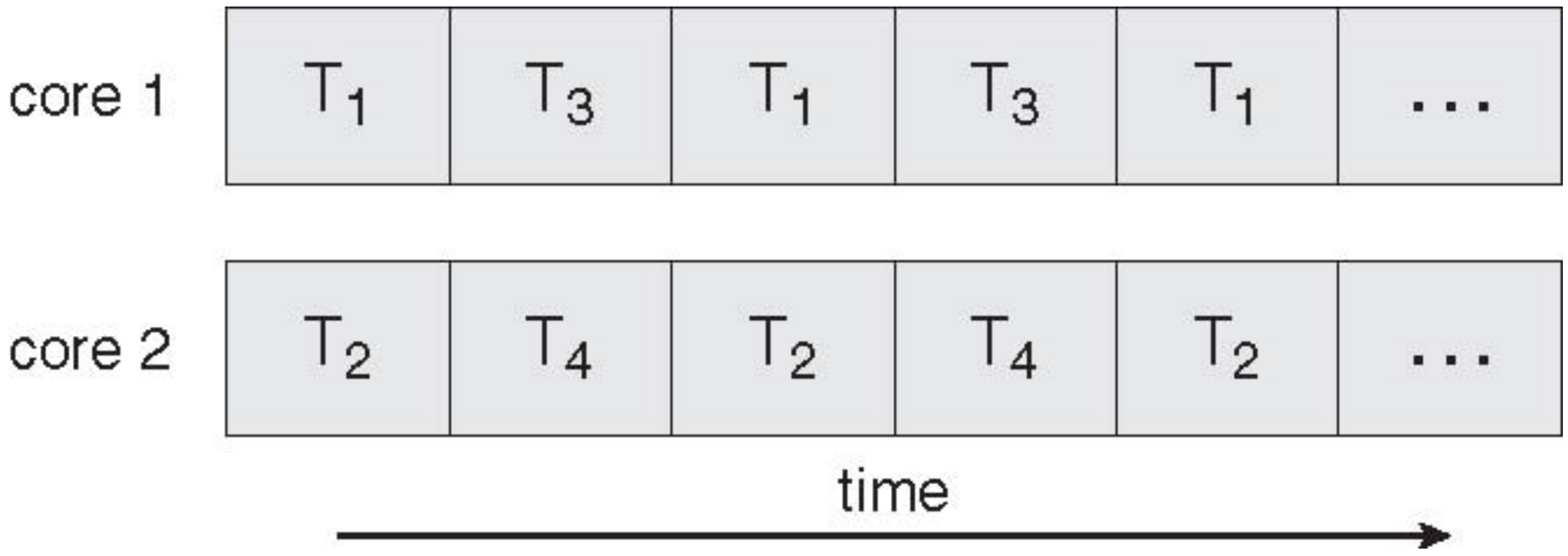
Thread and Process



Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



Implementing Threads

- Thread may be provided either at the user level, or by the kernel
  - **user threads** are supported above the kernel and managed without kernel support
    - three thread libraries: **POSIX Pthreads, Win32 threads**, and **Java threads**
  - **kernel threads** are supported and managed directly by the kernel
    - all contemporary OS supports kernel threads

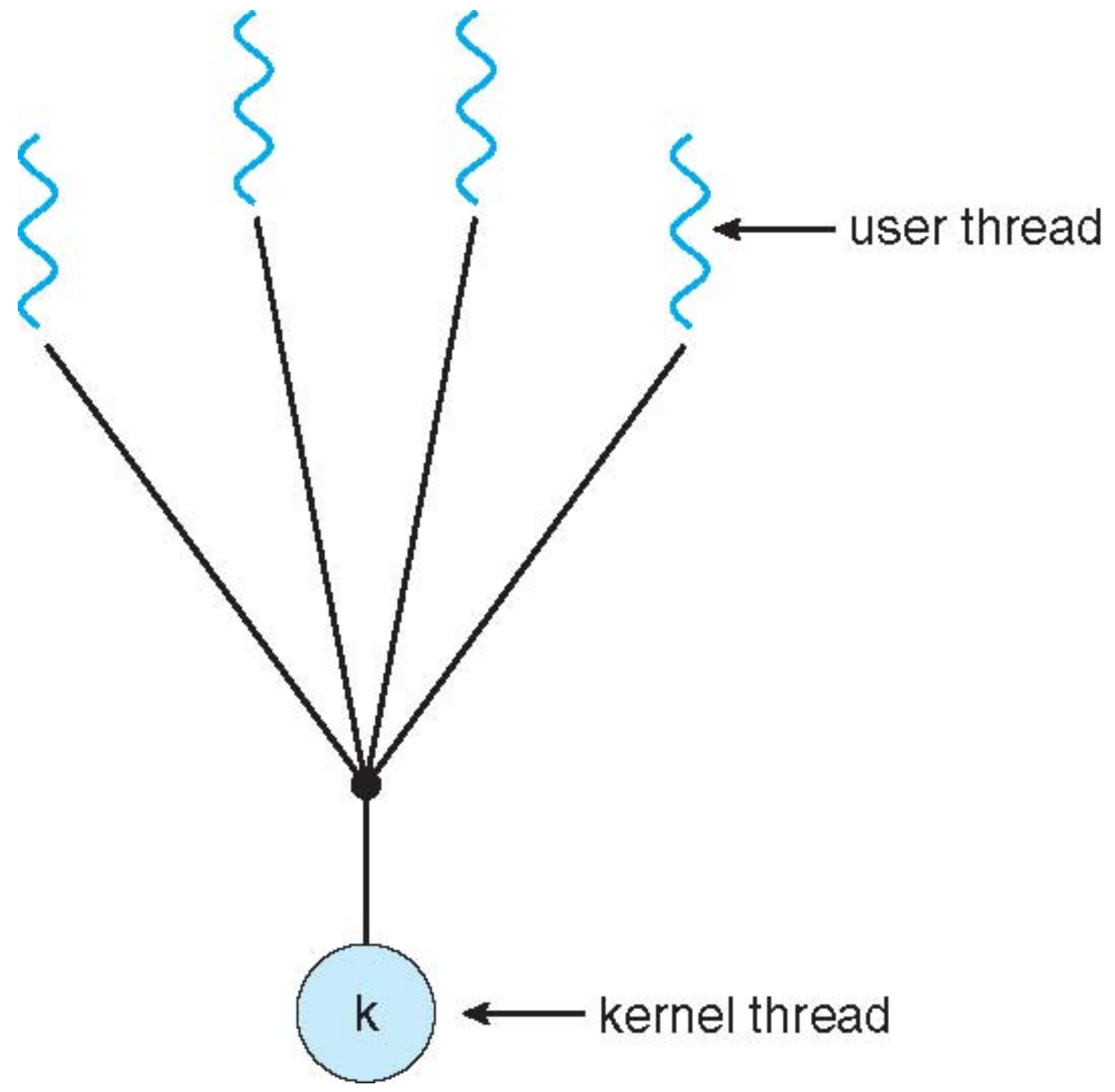
Multithreading Models

- A relationship **must exist** between user threads and kernel threads
  - Kernel threads are the real threads in the system, so for a user thread to make progress the user program has to have its scheduler take a user thread and then run it on a kernel thread.

Many-to-One

- Many user-level threads mapped to a single kernel thread
    - thread management is done by the thread library in **user space** (efficient)
    - entire process will block if a thread makes a blocking system call
        - convert blocking system call to non-blocking (e.g., select in Unix)?
    - multiple threads are unable to run in parallel on multi-processors
- Examples:
    - Solaris green threads

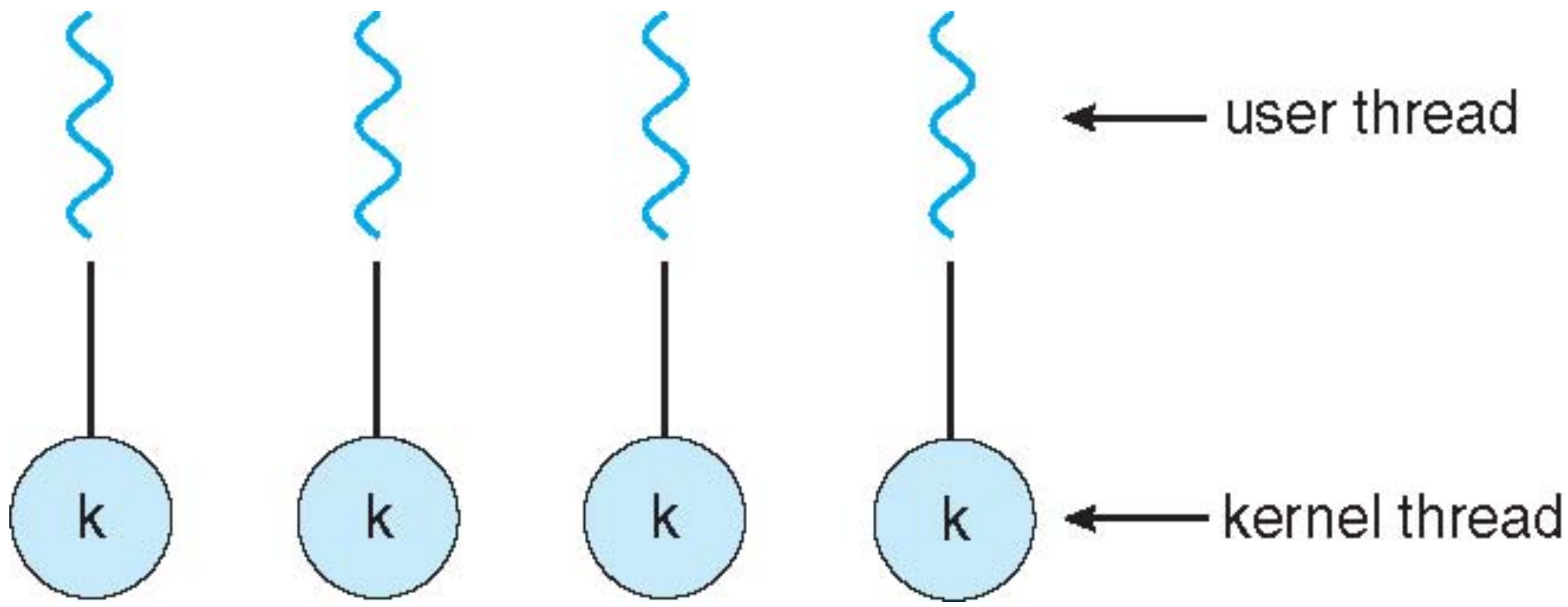
Many-to-One Model



One-to-One

- Each user-level thread maps to one kernel thread
    - it allows other threads to run when a thread blocks
    - multiple thread can run in parallel on multiprocessors
    - creating a user thread requires creating a corresponding kernel thread
        - it leads to overhead
    - most operating systems implementing this model limit the number of threads
- Examples
    - Windows NT/XP/2000
    - Linux

One-to-one Model



Many-to-Many Model

- Many user level threads are mapped to many kernel threads
  - it solves the shortcomings of 1:1 and m:1 model
  - developers can create as many user threads as necessary
  - corresponding kernel threads can run in parallel on a multiprocessor
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package

Semantics of Fork and Exec

- **Fork** duplicates the whole single-threaded process
- Does fork duplicate only the **calling** thread or **all** threads for multi-threaded process?
  - some UNIX systems have two versions of fork, one for each semantic
- **Exec** typically replaces the **entire** process, multithreaded or not
  - use “fork the calling thread” if calling exec soon after fork
- Which version of fork to use depends on the application
  - Exec is called immediately after forking: duplicating all threads is not necessary
  - Exec is not called: duplicating all threads

Pthreads

- A POSIX standard API for thread **creation** and **synchronization**
  - common in UNIX operating systems (Solaris, Linux, Mac OS X)
  - Pthread is a specification for thread behavior
    - implementation is up to developer of the library
    - e.g., Pthreads may be provided either as user-level or kernel-level

- **POSIX.1c**: Threads extensions (IEEE Std 1003.1c-1995)
  - Thread Creation, Control, and Cleanup
  - Thread Scheduling
  - Thread Synchronization
  - Signal Handling

How does Linux implement threads?

- User-level threads in Linux follow the open POSIX (Portable Operating System Interface for uniX) standard, designated as IEEE 1003. The user-level library (on Ubuntu, glibc.so) has an implementation of the POSIX API for threads.

- Threads exist in two separate execution spaces in Linux — in **user space** and the **kernel.**

  - User-space threads are created with the pthread library API (POSIX compliant).

  - In Linux, kernel threads are regarded as **"light-weight processes"**. An LWP is the unit of a basic execution context. Unlike other UNIX variants, including HP-UX and SunOS, there is no special treatment for threads. **A process or a thread in Linux is treated as a "task"**, and shares the same structure representation (list of struct task_structs).

  - These user-space threads are mapped to kernel threads.

How does Linux implement threads?

- For a set of user threads created in a user process, there is a set of corresponding LWPs in the kernel

```
os@os:~/os2018fall/code/4_thread/lwp1$ ./lwp1
LWP id is 20420
POSIX thread id is 0
```

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

int main()
{
    pthread_t tid = pthread_self();
    int sid = syscall(SYS_gettid);
    printf("LWP id is %dn", sid);
    printf("POSIX thread id is %dn", tid);
    return 0;
}
```

```
os@os:~$ ps -efL
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root         1     0     1  0    1 Oct13 ?        00:00:05 /sbin/init text
root         2     0     2  0    1 Oct13 ?        00:00:00 [kthreadd]
root         4     2     4  0    1 Oct13 ?        00:00:00 [kworker/0:0H]
root         6     2     6  0    1 Oct13 ?        00:00:00 [mm_percpu_wq]
root         7     2     7  0    1 Oct13 ?        00:00:00 [ksoftirqd/0]
root         8     2     8  0    1 Oct13 ?        00:00:02 [rcu_sched]
root         9     2     9  0    1 Oct13 ?        00:00:00 [rcu_bh]
root        10     2    10  0    1 Oct13 ?        00:00:00 [migration/0]
root        11     2    11  0    1 Oct13 ?        00:00:00 [watchdog/0]
root        12     2    12  0    1 Oct13 ?        00:00:00 [cpuhp/0]
root        13     2    13  0    1 Oct13 ?        00:00:00 [cpuhp/1]
root        14     2    14  0    1 Oct13 ?        00:00:00 [watchdog/1]
root        15     2    15  0    1 Oct13 ?        00:00:00 [migration/1]
root        16     2    16  0    1 Oct13 ?        00:00:00 [ksoftirqd/1]
root        18     2    18  0    1 Oct13 ?        00:00:00 [kworker/1:0H]

root       761     1   761  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   806  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   807  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   808  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   822  0    8 Oct13 ?        00:00:01 /usr/lib/snapd/snapd
root       761     1   823  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1   824  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root       761     1  4293  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
```

Clone system call

```c
casmlinkage int sys_clone(struct pt_regs regs)
{
        /* 注释中是i385下增加的代码, 其他体系结构无此定义
    unsigned long clone_flags;
    unsigned long newsp;

    clone_flags = regs.ebx;
    newsp = regs.ecx;*/
        if (!newsp)
            newsp = regs.esp;
        return do_fork(clone_flags, newsp, &regs, 0);
}
```

```c
asmlinkage long sys_vfork(struct pt_regs regs)
{
        return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.rsp, &regs, 0);
}
```

```c
asmlinkage long sys_fork(struct pt_regs regs)
{
        return do_fork(SIGCHLD, regs.rsp, &regs, 0);
}
```

05: CPU Scheduling

Basic Concepts

- Process execution consists of a cycle of CPU execution and I/O wait
    - **CPU burst** and **I/O burst** alternate
    - CPU burst distribution varies greatly from process to process, and from computer to computer, but follows similar curves
- Maximum CPU utilization obtained with **multiprogramming**
    - CPU scheduler selects another process when current one is in I/O burst

CPU Scheduler

- CPU scheduler selects from among the processes in **ready queue**, and allocates the CPU to one of them
- CPU scheduling decisions **may take place** when a process:
  - switches from **running to waiting state** (e.g., wait for I/O)
  - switches from **running to ready state** (e.g., when an interrupt occurs)
  - switches from **waiting to ready** (e.g., at completion of I/O)
  - **terminates**
- Scheduling under condition**1 and 4 only** is **nonpreemptive**
  - once the CPU has been allocated to a process, the process keeps it until terminates or waiting for I/O
  - also called **cooperative scheduling**
- **Preemptive scheduling** schedules process ***also*** in condition **2 and 3**

Scheduling Criteria

- **CPU utilization** : percentage of CPU being busy
- Throughput: # of processes that complete execution per time unit
- Turnaround time: the time to execute a particular process
  - from the time of *submission* to the time of *completion*
- **Waiting time**: the total time spent waiting in the *ready queue*
- Response time: the time it takes from when a request was submitted until the first response is produced
  - the time it takes to *start responding*

Scheduling Algorithms

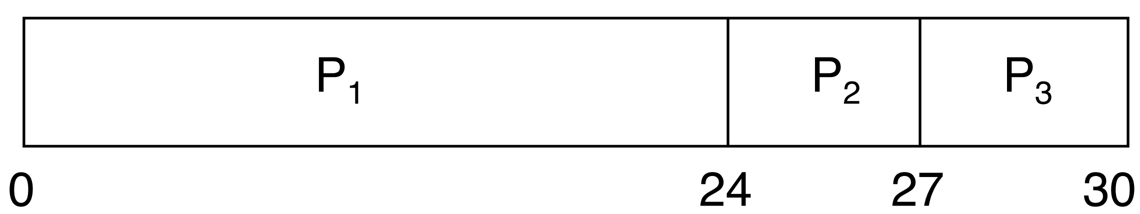
- First-come, first-served scheduling (FCFS)
- Shortest-job-first scheduling (SJF)
- Priority scheduling
- Round-robin scheduling (RR)
- Multilevel queue scheduling
- Multilevel feedback queue scheduling

First-Come, First-Served (FCFS) Scheduling

- Example processes:

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
- the Gantt Chart for the FCFS schedule is:



- **Waiting time** for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$, **average waiting time**: $(0 + 24 + 27)/3 = 17$

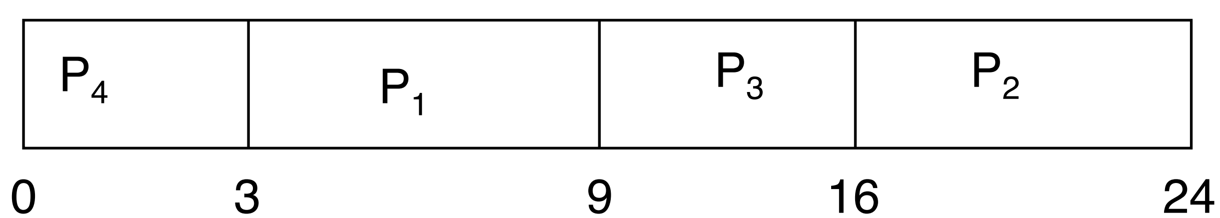
Shortest-Job-First Scheduling

- Associate with each process: the length of its next CPU burst
  - the process with the **smallest next CPU burst** is scheduled to run next
- SJF is **provably optimal**: it gives **minimum average waiting** time for a given set of processes
  - moving a short process before a long one decreases the overall waiting time
  - the difficulty is to know the length of the next CPU request
    - long-term scheduler can use the **user-provided processing time estimate**
    - short-term scheduler needs to **approximate** SFJ scheduling
- SJF can be **preemptive** or **nonpreemptive**
  - preemptive version is called **shortest-remaining-time-first**

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- **Average waiting time = (3 + 16 + 9 + 0) / 4 = 7**

Shortest-Remaining-Time-First

- SJF can be **preemptive**: **reschedule when a process arrives**

Process	**Arrival Time**	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

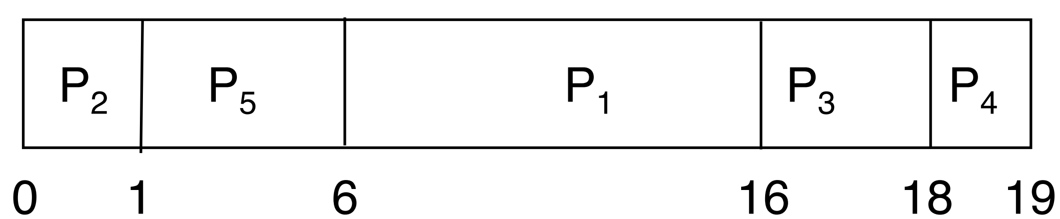
Priority Scheduling

- Priority scheduling selects the ready process with **highest priority**
  - a priority number is associated with each process, smaller integer, higher priority
  - the CPU is allocated to the process with the highest priority
  - SJF is special case of priority scheduling
    - priority is the inverse of predicted next CPU burst time
- Priority scheduling can be **preemptive** or **nonpreemptive**, similar to SJF
- **Starvation** is a problem: **low priority processes may never execute**
  - **Solution: aging — gradually increase priority of processes that wait for a long time**

Example of Priority Scheduling

ProcessA	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



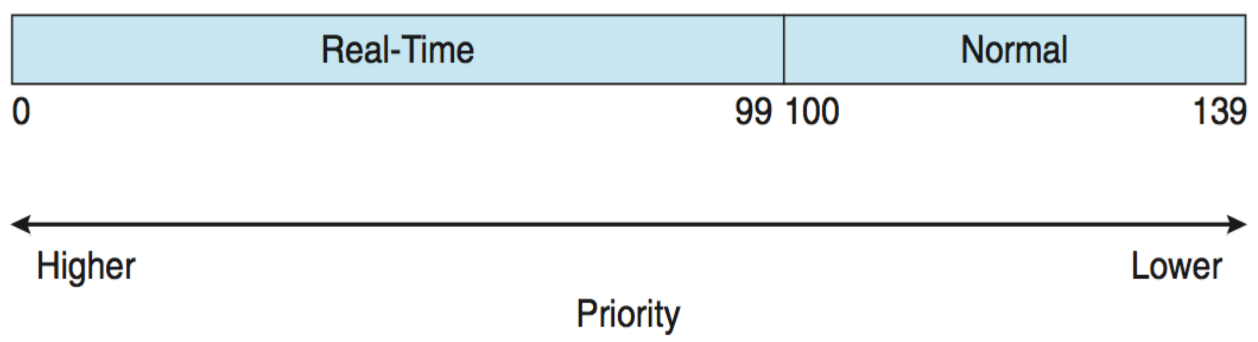
- Average waiting time = 8.2 msec

We use small number to denote high priority.

Round Robin (RR)

- Round-robin scheduling selects process in a **round-robin** fashion
  - each process gets a small unit of CPU time (time quantum, q)
    - q is too large ➙ FIFO, q is too small ➙ context switch overhead is high
    - a time quantum is generally 10 to 100 milliseconds

Linux



- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139
- rt_priority: only works for real time process.

06: Synchronization Tools

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in **data inconsistency**
  - data consistency requires orderly execution of cooperating processes

Uncontrolled Scheduling

- Counter = counter + 1

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

OS	Thread 1	Thread 2	PC	%eax	counter
			(after instruction)		
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

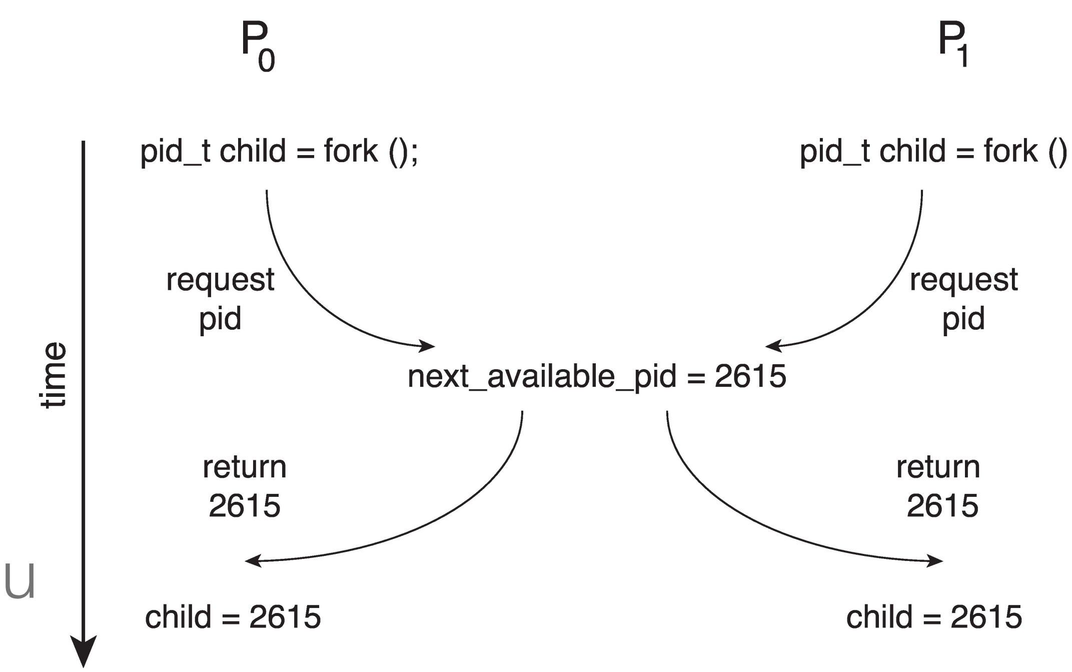
counter: 51 instead of 52!

Race Condition

- Several processes (or threads) access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race-condition**

Race Condition in Kernel

- Processes P0 and P1 are creating child processes using the fork() system call
- Race condition on kernel variable ***next_available_pid*** which represents the next available process identifier (pid)



- Unless there is mutu assigned to two different processes!
- **Even if the kernel is non-preemptive, race condition can still exist in user space!**

Critical Section

- General structure of process p_i is

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to Critical-Section: Three Requirements

- **Mutual Exclusion**
  - only one process can execute in the critical section
- **Progress**
- **Bounded waiting**
  - it prevents **starvation**

Hardware Instructions

- Special hardware instructions that allow us to either test-and-modify the content of a word, or two swap the contents of two words atomically (uninterruptibly.)

- **Test-and-Set** instruction

- **Compare-and-Swap** instruction

Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is **mutex lock**
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire() and release()** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
- This lock therefore called a spinlock

Mutex Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Lock Definitions

- ```
  acquire() {
     while (!available)
        ; /* busy wait */
     available = false;;
  }
  ```

- ```
  release() {
     available = true;
  }
  ```

- **These two functions must be implemented atomically.**
- Both test-and-set and compare-and-swap can be used to implement these functions.

Semaphore

- **Semaphore** S is an integer variable
  - e.g., to represent *how many units of a particular resource is available*
- It can only be updated with two **atomic** operations: **wait** and **signal**
  - **spin lock** can be used to guarantee atomicity of wait and signal
  - originally called P and V (Dutch)
  - a simple implementation with busy wait can be:

```
wait(s)
{
    while (s <= 0) ; //busy wait
    s--;
```

```
signal(s)
{
    s++;
}
```