Final Review 02

Operating Systems

Wenbo Shen

Acknowledgement: Some slides are based on Yajin Zhou@ZJU's slides.

Summary

- Computer architecture
- OS introduction
- OS structures
- Processes
- IPC
- Thread
- Scheduling
- Synchronization
- Deadlock

Summary

- Memory – segmentation
- Memory – paging
- Virtual memory
- Virtual memory – Linux
- Mass storage
- IO
- FS interface
- FS implementation
- FS in practice

04: Thread

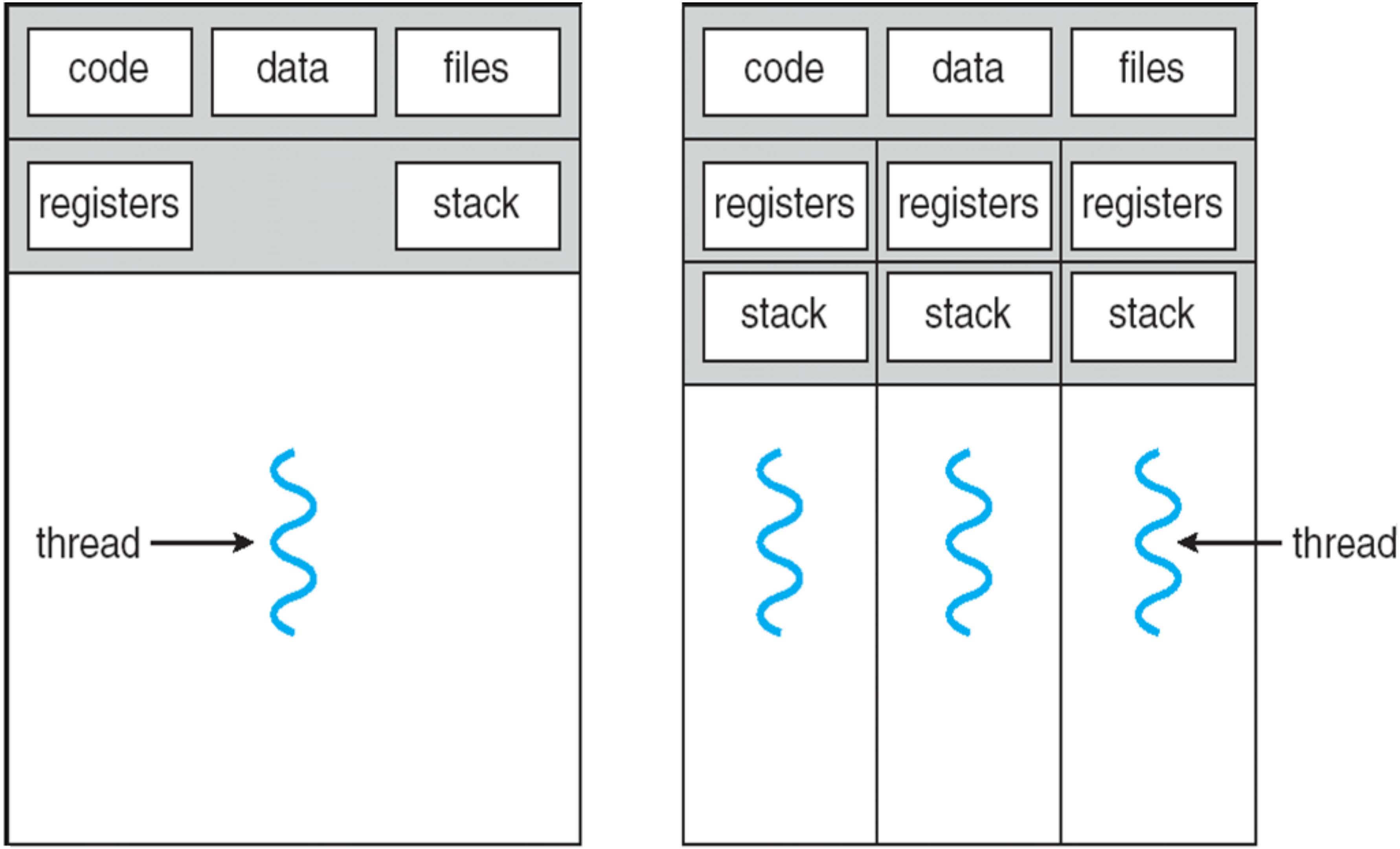
Motivation

- 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 Definition

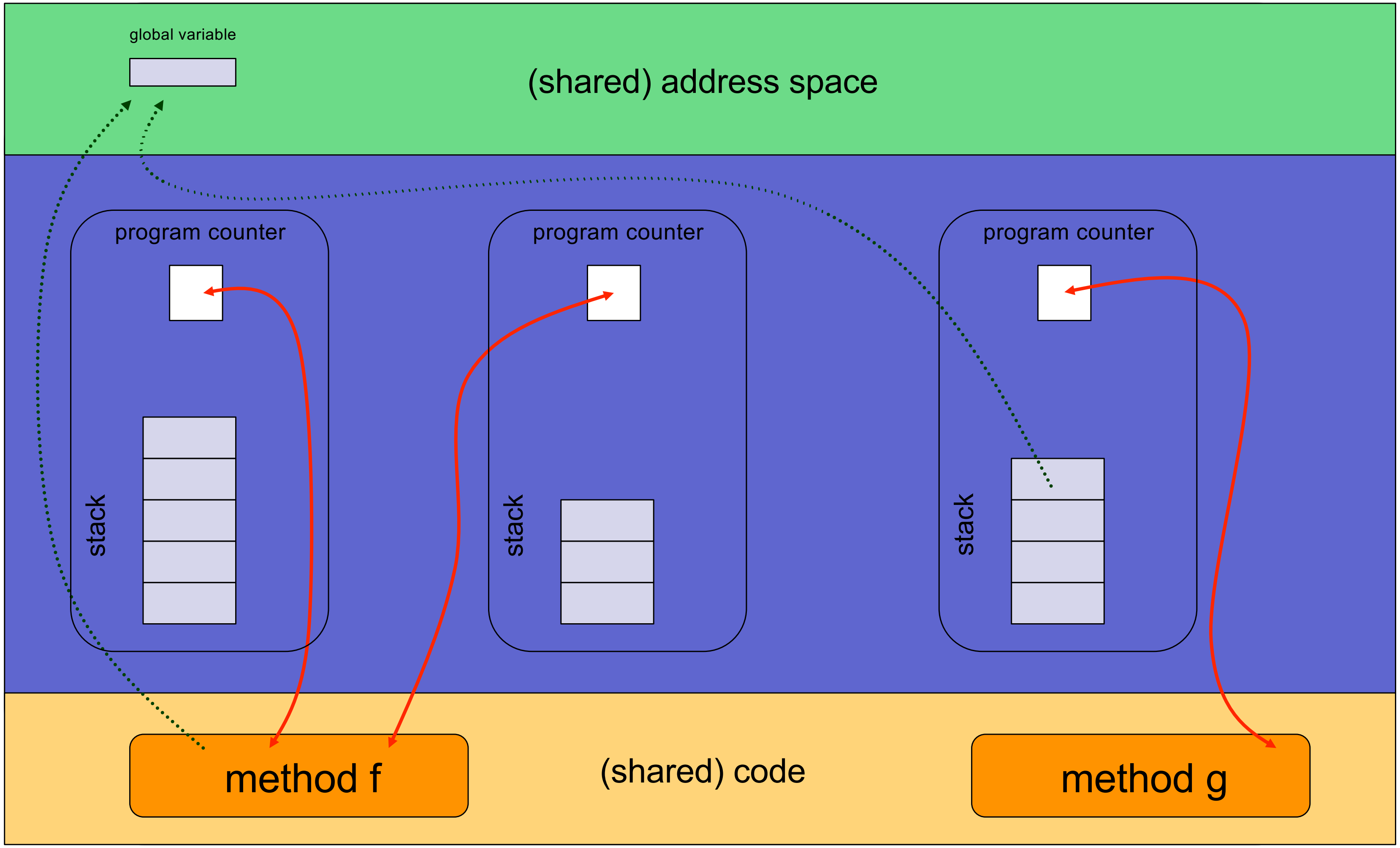
- A thread is a basic unit of execution within a process
- Each thread has its own
  - thread ID
  - program counter
  - register set
  - Stack
- It shares the following with other threads within the same process
  - code section
  - data section
  - the heap (dynamically allocated memory)
  - open files and signals
- **Concurrency**: A multi-threaded process can do multiple things at once

The Typical Figure



single-threaded process multithreaded process

Thread and Process



Advantages of Threads

- Economy:
  - Creating a thread is cheap
    - Much cheaper than creating a process
      - Code, data and heap are already in memory
  - Context-switching between threads is cheap
    - Much cheaper than between processes
      - No cache flush
- Resource Sharing:
  - Threads naturally share memory
    - With processes you have to use possibly complicated IPC (e.g., Shared Memory Segments)
    - IPC is not needed
  - Having concurrent activities in the same address space is very powerful
    - But fraught with danger

Advantages of Threads?

- Responsiveness
  - A program that has concurrent activities is more responsive
    - While one thread blocks waiting for some event, another can do something
    - e.g. Spawn a thread to answer a client request in a client-server implementation
  - This is true of processes as well, but with threads we have better sharing and economy
- Scalability
  - Running multiple “threads” at once uses the machine more effectively
    - e.g., on a multi-core machine
  - This is true of processes as well, but with threads we have better sharing and economy

Drawbacks of Threads

- Weak isolation between threads: If one thread fails (e.g., a segfault), then the process fails
  - And therefore the whole program
- Threads may be more memory-constrained than processes
  - Due to OS limitation of the address space size of a single process
  - Not a problem any more on 64-bit architecture
- Threads do not benefit from memory protection
  - Concurrent programming with Threads is hard
    - But so is it with Processes and Shared Memory Segments

Implementing Threads

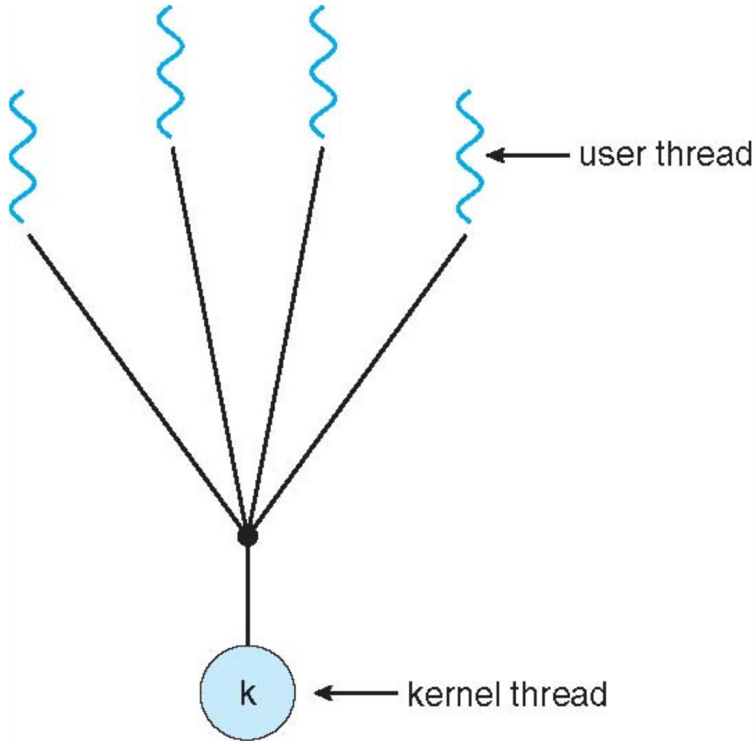
- Thread may be provided either at the user level, or by the kernel
  - User threads are supported above the kernel and managed without kernel support
    - Three thread libraries: POSIX pthreads, win32 threads, and java threads
  - Kernel threads are supported and managed directly by the kernel
    - 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.

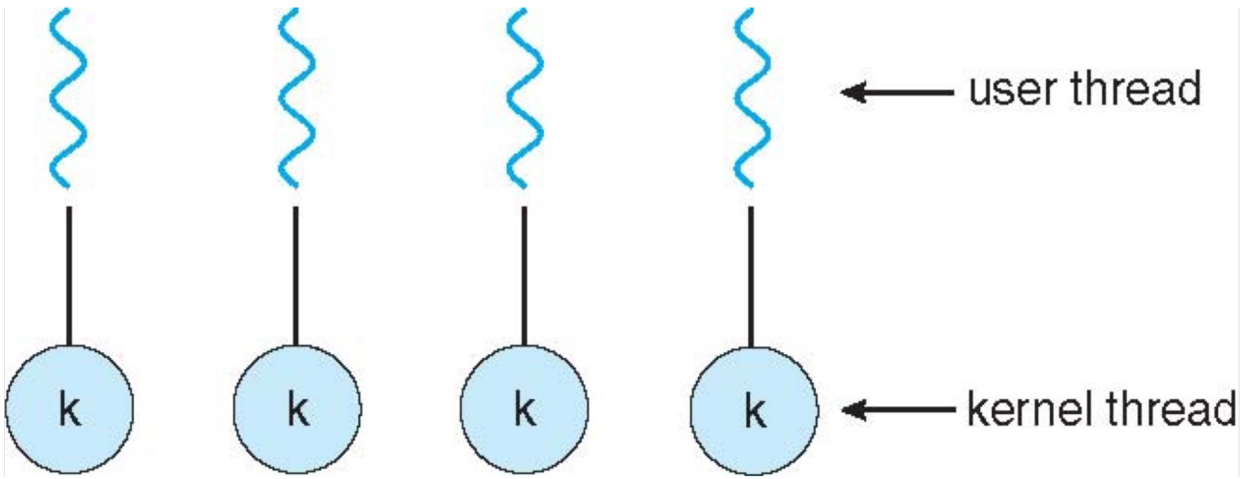
Many-to-One

- 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



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

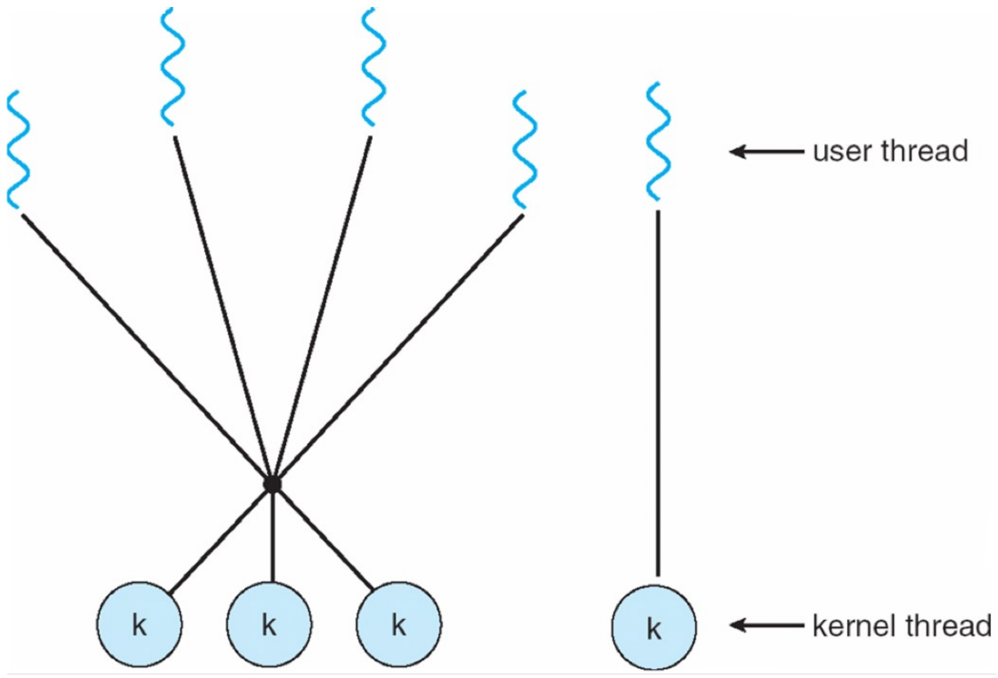


Many-to-Many Model

- 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

Two-level Model

- Similar to many-to-many model, except that it allows a user thread to be **bound** to kernel thread



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

Linux Threads

- Linux does not distinguish between PCB and TCB
  - Kernel data structure: task_struct

```

struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;

        /*
         * This begins the randomizable portion of task_struct. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

        void                            *stack;
        atomic_t                        usage;
        /* Per task flags (PF_*), defined further below: */
        unsigned int                    flags;
        unsigned int                    ptrace;
```

Linux Threads

- In Linux, a thread is also called a light-weight process (LWP)
- The clone() syscall is used to create a thread or a process
  - Shares execution context with its parent
  - pthread library uses clone() to implement threads. Refer to ./nptl/sysdeps/pthread/createthread.c

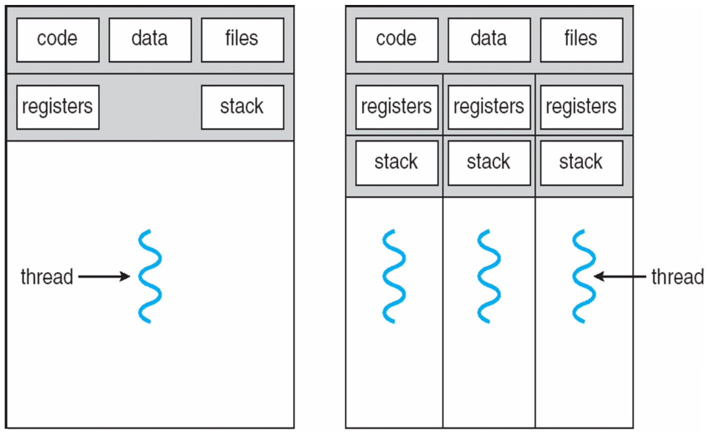
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Linux Threads

- Single-threaded process vs multi-threaded process

```
wenbo@wenbo-desktop:~/KERNEL/linux.git$ ps -eLf
UID          PID   PPID    LWP  C NLWP STIME TTY          TIME CMD
root           1      0      1  0    1 3月11 ?        00:00:19 /sbin/init splash
root           2      0      2  0    1 3月11 ?        00:00:00 [kthreadd]
root           4      2      4  0    1 3月11 ?        00:00:00 [kworker/0:0H]
root           6      2      6  0    1 3月11 ?        00:00:00 [mm_percpu_wq]
root           7      2      7  0    1 3月11 ?        00:00:00 [ksoftirqd/0]
root           8      2      8  0    1 3月11 ?        00:00:31 [rcu_sched]
root           9      2      9  0    1 3月11 ?        00:00:00 [rcu_bh]
root          10      2     10  0    1 3月11 ?        00:00:00 [migration/0]
root          11      2     11  0    1 3月11 ?        00:00:00 [watchdog/0]

root         704      1    704  0    1 3月11 ?        00:00:00 /usr/sbin/cron -f
root         718      1    718  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root         718      1    882  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root         718      1    883  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root         718      1    884  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root         718      1    885  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root         718      1    917  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root         718      1    921  0   16 3月11 ?        00:00:01 /usr/lib/snapd/snapd
root         718      1    922  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root         718      1    923  0   16 3月11 ?        00:00:01 /usr/lib/snapd/snapd
root         718      1    924  0   16 3月11 ?        00:00:01 /usr/lib/snapd/snapd
```



single-threaded process   multithreaded process

Linux Threads

- Single-threaded process vs multi-threaded process

```
wenbo@wenbo-desktop:~/KERNEL/linux.git$ ps -eLf
UID        PID  PPID   LWP  C NLWP STIME TTY          TIME CMD
root         1     0     1  0    1 3月11 ?        00:00:19 /sbin/init splash
root         2     0     2  0    1 3月11 ?        00:00:00 [kthreadd]
root         4     2     4  0    1 3月11 ?        00:00:00 [kworker/0:0H]
root         6     2     6  0    1 3月11 ?        00:00:00 [mm_percpu_wq]
root         7     2     7  0    1 3月11 ?        00:00:00 [ksoftirqd/0]
root         8     2     8  0    1 3月11 ?        00:00:31 [rcu_sched]
root         9     2     9  0    1 3月11 ?        00:00:00 [rcu_bh]
root        10     2    10  0    1 3月11 ?        00:00:00 [migration/0]
root        11     2    11  0    1 3月11 ?        00:00:00 [watchdog/0]

root       704     1   704  0    1 3月11 ?        00:00:00 /usr/sbin/cron -f
root       718     1   718  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root       718     1   882  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root       718     1   883  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root       718     1   884  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root       718     1   885  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root       718     1   917  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root       718     1   921  0   16 3月11 ?        00:00:01 /usr/lib/snapd/snapd
root       718     1   922  0   16 3月11 ?        00:00:00 /usr/lib/snapd/snapd
root       718     1   923  0   16 3月11 ?        00:00:01 /usr/lib/snapd/snapd
root       718     1   924  0   16 3月11 ?        00:00:01 /usr/lib/snapd/snapd
```

```
	/* PID/PID hash table linkage. */
	struct pid			*thread_pid;
	struct hlist_node		pid_links[PIDTYP
	struct list_head		thread_group;
	struct list_head		thread_node;

	struct completion		*vfork_done;

	/* CLONE_CHILD_SETTID: */
	int __user			*set_child_tid;
```

Threads with Process – What is shared

```
static void traversal_thread_group(struct task_struct * tsk){
        struct task_struct * curr_thread = NULL;
        unsigned long tg_offset = offsetof(struct task_struct, thread_group);

        curr_thread = (struct task_struct *) (((unsigned long)tsk->thread_group.next) - tg_offset);
        while (curr_thread != tsk){
                printk("\t\tTHREAD TSK=%llx\tPID=%d\tSTACK=%llx \tCOMM=%s\tMM=%llx\tACTIVE_MM=%llx\n",
                                (u64)curr_thread, curr_thread->pid, (u64)curr_thread->stack,
                                curr_thread->comm, (u64)curr_thread->mm, (u64)curr_thread->active_mm);
                curr_thread = (struct task_struct *) (((unsigned long)curr_thread->thread_group.next) - tg_offset);
        }
}

static void traversal_process(void) {
        struct task_struct * tsk = NULL;

        traversal_thread_group(&init_task);
        for_each_process(tsk){
                printk("PROCESS\tTHREAD TSK=%llx\tPID=%d\tSTACK=%llx \tCOMM=%s\tMM=%llx\tACTIVE_MM=%llx\n",
                                (u64)tsk, tsk->pid, (u64)tsk->stack, tsk->comm,
                                (u64)tsk->mm, (u64)tsk->active_mm);
                traversal_thread_group(tsk);
        }
}
```

Threads with Process – What is shared

```c
static void traversal_thread_group(struct task_struct * tsk){
        struct task_struct * curr_thread = NULL;
        unsigned long tg_offset = offsetof(struct task_struct, thread_group);

        curr_thread = (struct task_struct *) (((unsigned long)tsk->thread_group.next) - tg_offset);
        while (curr_thread != tsk){
                printk("\t\tTHREAD TSK=%llx\tPID=%d\tSTACK=%llx \tCOMM=%s\tMM=%llx\tACTIVE_MM=%llx\n",
                                (u64)curr_thread, curr_thread->pid, (u64)curr_thread->stack,
                                curr_thread->comm, (u64)curr_thread->mm, (u64)curr_thread->active_mm);
                curr_thread = (struct task_struct *) (((unsigned long)curr_thread->thread_group.next) - tg_offset);
        }
}

static void traversal_process(void) {
        struct task_struct * tsk = NULL;

        traversal_thread_group(&init_task);
        for_each_process(tsk){
                printk("PROCESS\tTHREAD TSK=%llx\tPID=%d\tSTACK=%llx \tCOMM=%s\tMM=%llx\tACTIVE_MM=%llx\n",
                                (u64)tsk, tsk->pid, (u64)tsk->stack, tsk->comm,
                                (u64)tsk->mm, (u64)tsk->active_mm);
                traversal_thread_group(tsk);
        }
}
```

```
PROCESS  THREAD TSK=ffff8c4c4bf3c5c0    PID=718 STACK=ffff985c82268000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46d52e80    PID=882 STACK=ffff985c82390000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46d545c0    PID=883 STACK=ffff985c822e8000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c491b45c0    PID=884 STACK=ffff985c8218c000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4beb1740    PID=885 STACK=ffff985c821ec000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4ae1ae80    PID=917 STACK=ffff985c823c8000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b562e80    PID=921 STACK=ffff985c82418000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c48340000    PID=922 STACK=ffff985c823b0000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c472bae80    PID=923 STACK=ffff985c821f4000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b5945c0    PID=924 STACK=ffff985c81fa8000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46775d00    PID=925 STACK=ffff985c822a8000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b692e80    PID=973 STACK=ffff985c82438000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b78ae80    PID=974 STACK=ffff985c823c0000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46e1dd00    PID=975 STACK=ffff985c824b8000  COMM=snapd     MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
```

Threads within Process – What is shared

```
PROCESS  THREAD TSK=ffff8c4c4bf3c5c0     PID=718 STACK=ffff985c82268000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46d52e80     PID=882 STACK=ffff985c82390000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46d545c0     PID=883 STACK=ffff985c822e8000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c491b45c0     PID=884 STACK=ffff985c8218c000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4beb1740     PID=885 STACK=ffff985c821ec000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4ae1ae80     PID=917 STACK=ffff985c823c8000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b562e80     PID=921 STACK=ffff985c82418000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c48340000     PID=922 STACK=ffff985c823b0000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c472bae80     PID=923 STACK=ffff985c821f4000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b5945c0     PID=924 STACK=ffff985c81fa8000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46775d00     PID=925 STACK=ffff985c822a8000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b692e80     PID=973 STACK=ffff985c82438000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c4b78ae80     PID=974 STACK=ffff985c823c0000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
         THREAD TSK=ffff8c4c46e1dd00     PID=975 STACK=ffff985c824b8000  COMM=snapd   MM=ffff8c4c46400840     ACTIVE_MM=ffff8c4c46400840
```

task_struct pid stack comm

mm_struct

Not Shared

Shar
ed

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
  - Rationale: non-CPU-intensive jobs should really get the CPU quickly on the rare occasions they need them, because they could be interactive processes
  - Maximum CPU utilization obtained with multiprogramming

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**
  - Preemptive scheduling needs hardware support such as a timer
  - Synchronization primitives are necessary
- Context switch can only happen in kernel node, so is preemption
  - User space processes need to trap to kernel mode to do context switch.

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

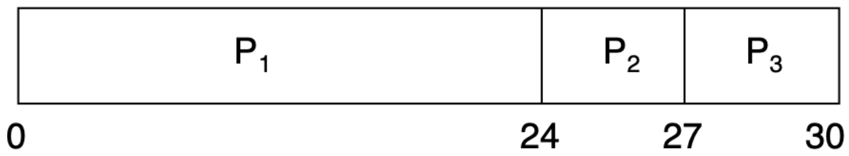
- First-come, first-served scheduling (FCFS)
- Shortest-job-first scheduling (SJF)
- Priority scheduling
- Round-robin scheduling (RR)
- Multilevel queue scheduling
- Multilevel feedback queue scheduling

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

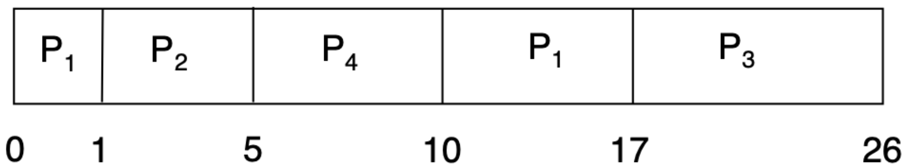
- **Average waiting time = (3 + 16 + 9 + 0) / 4 = 7**

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



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

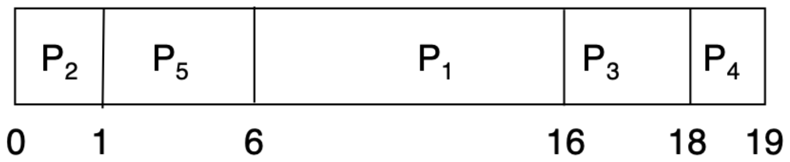
Priority Scheduling

- 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

Process	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

06&07: Synchronization

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	(after instruction) PC	%eax	counter
	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**

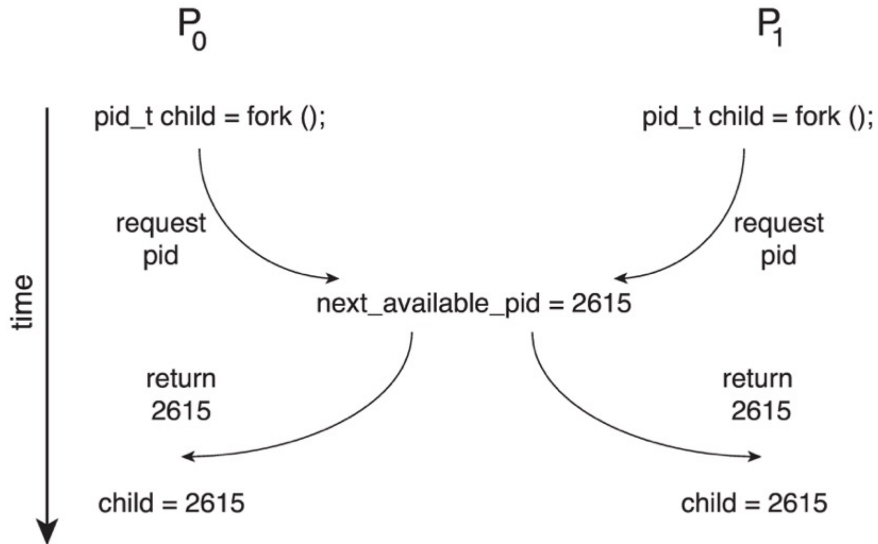
counter: 51 instead of 52!

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 mutual exclusion, the same pid could be 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);
```

Critical-Section Handling in OS

- Single-core system: preventing interrupts
- Multiple-processor: preventing interrupts are not feasible
- Two approaches depending on if kernel is ***preemptive or non-preemptive***
  - Preemptive – allows preemption of process when running in kernel mode
  - Non-preemptive – runs until **exits kernel mode, blocks, or voluntarily yields CPU**
    - **Essentially free of race conditions *in kernel mode, but NOT for user space!!***

Solution to Critical-Section: Three Requirements

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

Peterson’s Solution

- Peterson’s solution solves **two-processes** synchronization
- **It’s a software based-solution**
- It assumes that LOAD and STORE are **atomic**
  - **atomic**: execution cannot be interrupted
- The two processes share two variables
  - int **turn**: whose turn it is to enter the critical section
  - Boolean **flag[2]**: whether a process is ready to enter the critical section

Peterson's Solution

```
flag[0] = FALSE;
flag[1] = FALSE;
```

- P_0:

Mark self ready

Assert the other one

```
do {
    flag[0] = TRUE;
    turn = 1;
    while (flag[1] && turn == 1);
    critical section
    flag[0] = FALSE;
    remainder section
} while (TRUE);
```

- P_1:

```
do {
    flag[1] = TRUE;
    turn = 0;
    while (flag[0] && turn == 0);
    critical section
    flag[1] = FALSE;
    remainder section
} while (TRUE);
```

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

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

```
bool locked = false;

acquire() {
    while (compare_and_swap(&locked, false, true))
        ; //busy waiting
}

release() {
    locked = false;
}
```

Semaphore

- **Semaphore** S is an integer variable
  - e.g., to represent *how many units of a particular resource is available*
  - *For resource sharing purpose*
- 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++;
}
```

Semaphore

- Associate a waiting queue with each semaphore
  - place the process on the waiting queue if **wait** cannot return immediately
  - wake up a process in the waiting queue in **signal**
- There is no need to **busy wait** in critical section
- Note: wait and signal must still be atomic

Semaphore

```
wait(semaphore *S) {
     S->value--;
     if (S->value < 0) {
          add this process to S->list;
          block();
     }
}

signal(semaphore *S) {
     S->value++;
     if (S->value <= 0) {
          remove a proc.P from S->list;
          wakeup(P);
     }
}
```

Suppose the init value s->value = 5
And now, if s->value = -3, what does it mean?

Busy waiting time - Mutex

- Mutex busy waiting time
  - From acquire to release

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

busy waiting

- What if the critical section is long?
  - A huge waste of CPU time

Busy waiting time - Semaphore

```
Semaphore sem;    //  initialized to 1
do {
    wait (sem);                busy waiting
    critical section
    signal (sem);              busy waiting
    remainder section
} while (TRUE);      //while loop but not busy waiting
```

- No busy waiting on critical section
- Still has the busy waiting on *wait* and *signal*
  - But waiting is much shorter

Bounded-Buffer Problem

- Two processes, the producer and the consumer share **n** buffers
  - the producer generates data, puts it into the buffer
  - the consumer consumes data by removing it from the buffer
- The problem is to make sure:
  - **the producer won’t try to add data into the buffer if it is full**
  - **the consumer won’t try to remove data from an empty buffer**
  - also call producer-consumer problem

Bounded-Buffer Problem

- Solution:
  - n buffers, each can hold one item
  - semaphore mutex initialized to the value 1
  - semaphore full-slots initialized to the value 0
  - semaphore empty-slots initialized to the value N

Bounded-Buffer Problem

- The producer process:

```
  do {
 //produce an item

   …

   wait(empty-slots);

   wait(mutex);

   //add the item to the  buffer

   …

   signal(mutex);

   signal(full-slots);

  } while (TRUE)
```

Bounded Buffer Problem

- The consumer process:

```
do {
 wait(full-slots);
 wait(mutex);
 //remove an item from  buffer
 …
 signal(mutex);
 signal(empty-slots);
 //consume the item
 …
} while (TRUE);
```

Takeaway

- Whole slides