Processes and Threads

Week 05 – Lecture 2

Team

Instructors

Giancarlo Succi

Joseph Brown

Teaching Assistants

Vladimir Ivanov

Stanislav Litvinov

Alexey Reznik

Munir Makhmutov

Hamna Aslam

Sources

- These slides have been adapted from the original slides of the adopted book:
  - Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
    Prentice-Hall, Inc.

  and customized for the needs of this course.
- Additional input for the slides are detailed later

Introduction to Scheduling (1)

- When a computer is multiprogrammed, it frequently has multiple processes or threads competing for the CPU at the same time
- If only one CPU is available, a choice has to be made which process to run next
- The part of the OS that makes the choice is called the **scheduler**, and the algorithm it uses is called the **scheduling algorithm**

Introduction to Scheduling (2)

- Batch systems: just run the next job on the tape
- Multiprogramming systems: multiple users are waiting for service and CPU time is a scarce resource, so the scheduling algorithm became more complex
- Personal computers: users are usually working in one application at a time and CPUs became much faster. As a consequence, scheduling does not matter much on simple PCs

Introduction to Scheduling (3)

- Networked servers: multiple processes often do compete for the CPU, so scheduling matters again
- Mobile devices: since battery lifetime is one of the most important constraints on these devices, some schedulers try to optimize the power consumption
- **Process switching is expensive**, so having a good scheduler is critical to promote performance

Introduction to Scheduling
Process Behavior (1)

- Nearly all processes alternate bursts of computing with (disk or network) I/O requests (Fig. 2-39)
- **Compute-bound or CPU-bound processes**: the processes that spend most of their time computing (Fig. 2-39a)
- **I/O-bound processes**: the processes that spend most of their time waiting for I/O (Fig. 2-39b)
- As CPUs get faster, processes tend to get more I/O-bound because CPUs are improving much faster than disks

Introduction to Scheduling Process Behavior (2)

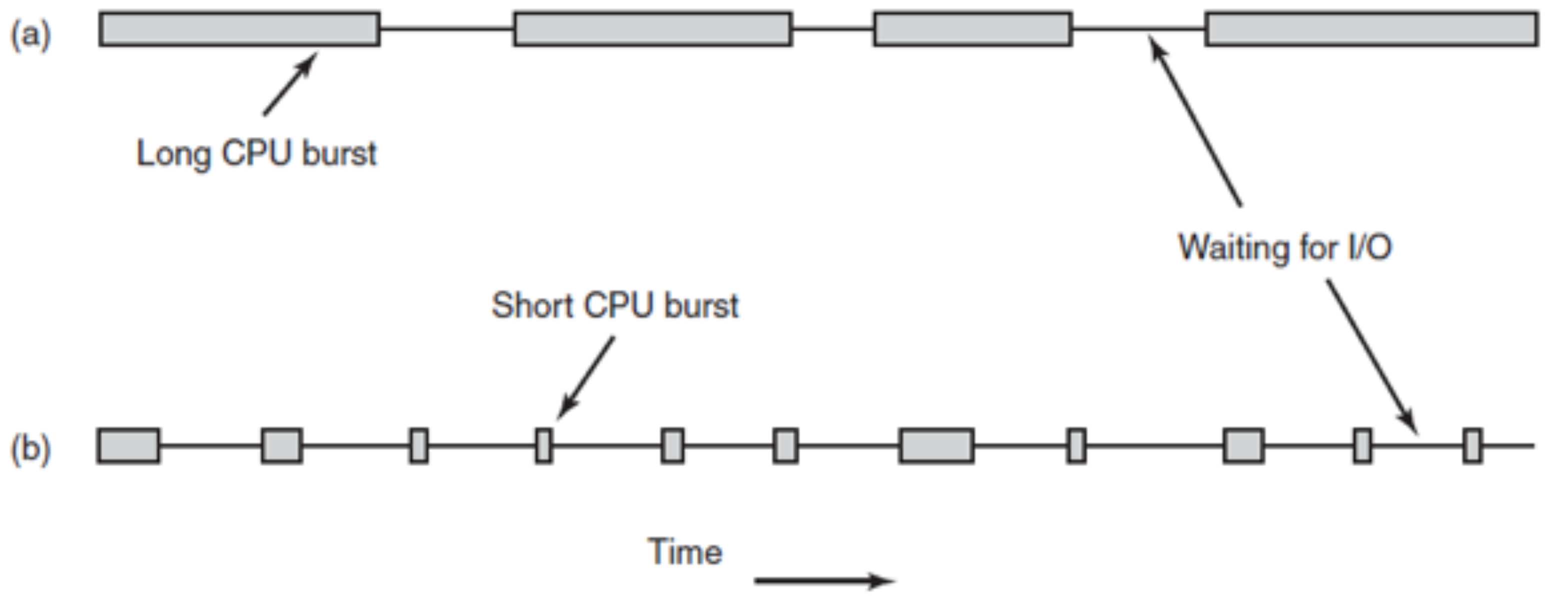


Figure 2-39. Bursts of CPU usage alternate with periods of waiting for I/O. (a) A CPU-bound process. (b) An I/O-bound process.

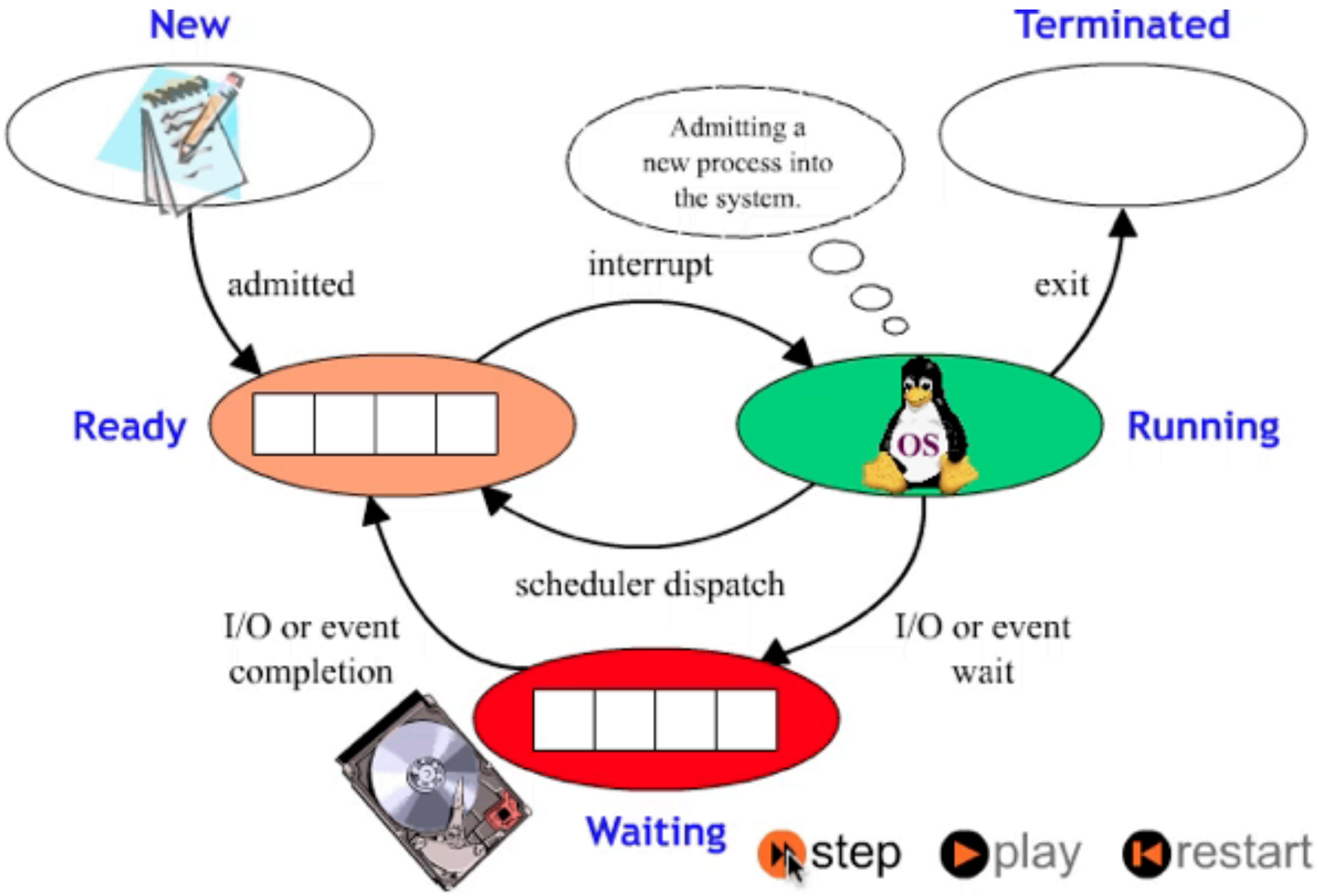
When To Schedule (1)

- When to make scheduling decisions?
  - **When a new process is created.** Both parent and child processes are in ready state and the scheduler can legitimately choose to run either the parent or the child next
  - **When a process exits.** That process can no longer run, so some other process must be chosen from the set of ready processes
  - **When a process blocks on I/O or on a semaphore** – another process has to be selected to run
  - **When an I/O interrupt occurs**, a scheduling decision may be made. Scheduler will decide whether to run the process that was waiting for the interrupt, the process that was running at the time of the interrupt, or some third process

When To Schedule (2)

- A scheduling decision can be made at each clock interrupt or at every k_{th} clock interrupt
- Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts:
  - A **nonpreemptive** scheduling algorithm picks a process to run and then just lets it run until it blocks (either on I/O or waiting for another process) or voluntarily releases the CPU. No scheduling decisions are made during clock interrupts
  - A **preemptive** scheduling algorithm picks a process and lets it run for a maximum of some fixed time. If it is still running at the end of the time interval, it is suspended and the scheduler picks another process to run (if one is available)

Scheduling Example



Categories of Scheduling Algorithms (1)

- In different environments different scheduling algorithms are needed
- What the scheduler should optimize for is not the same in all systems. Three environments worth distinguishing are
  - Batch
  - Interactive
  - Real-time

Categories of Scheduling Algorithms (2)

- Batch systems:
  - in use in the business world for doing payroll, inventory, accounts receivable, accounts payable, interest calculation (at banks), claims processing (at insurance companies), and other periodic tasks.
  - there are no users impatiently waiting at their terminals for a quick response to a short request
  - nonpreemptive algorithms, or preemptive algorithms with long time periods for each process, are often acceptable. This approach reduces process switches and thus improves performance

Categories of Scheduling Algorithms (3)

- Interactive environment:
  - preemption is essential to keep one process from hogging the CPU and denying service to the others
  - even if no process intentionally ran forever, one process might shut out all the others indefinitely due to a program bug
  - preemption is needed to prevent this behavior.
  - servers also fall into this category, since they normally serve multiple (remote) users

Categories of Scheduling Algorithms (4)

- Real-time environment:
  - preemption is sometimes not needed because the processes know that they may not run for long periods of time and usually do their work and block quickly
  - real-time systems run only programs that are intended to further the application at hand, as opposed to the interactive systems that are general purpose and may run arbitrary programs that are not cooperative and even possibly malicious

Scheduling Algorithm Goals (1)

- All systems:
  - **Fairness** – giving each process a fair share of the CPU. Comparable processes should get comparable service
  - **Policy enforcement** – seeing that stated policy is carried out
  - **Balance** – keeping all parts of the system busy. If the CPU and all the I/O devices can be kept running all the time, more work gets done per second than if some of the components are idle

Scheduling Algorithm Goals (2)

- Batch systems:
  - **Throughput** – maximize jobs per hour
  - **Turnaround time** – minimize time between submission and termination
  - **CPU utilization** – keep the CPU busy all the time
- Maximizing throughput does not necessarily mean minimizing turnaround time. If there is a mix of short and long jobs and the scheduler always picks the short jobs first, turnaround time for long jobs will be too long

Scheduling Algorithm Goals (3)

- Interactive systems:
  - **Response time** – respond to requests quickly. A user request to start a program or open a file should take precedence over the background work
  - **Proportionality** – meet users' expectations. In users' perception, uploading a large file should take some time, but breaking a connection to a server should not. Scheduler's job is to make sure that users' expectations are met

Scheduling Algorithm Goals (4)

- Real-time systems:
  - **Meeting deadlines** – avoid losing data. Components of a real-time system have deadlines and if one component misses the deadline, another one may lose data that should have been otherwise consumed by the first one
  - **Predictability** – avoid quality degradation in multimedia systems. For example, if the audio process runs too erratically, the sound quality will deteriorate rapidly and the human's ear will detect it

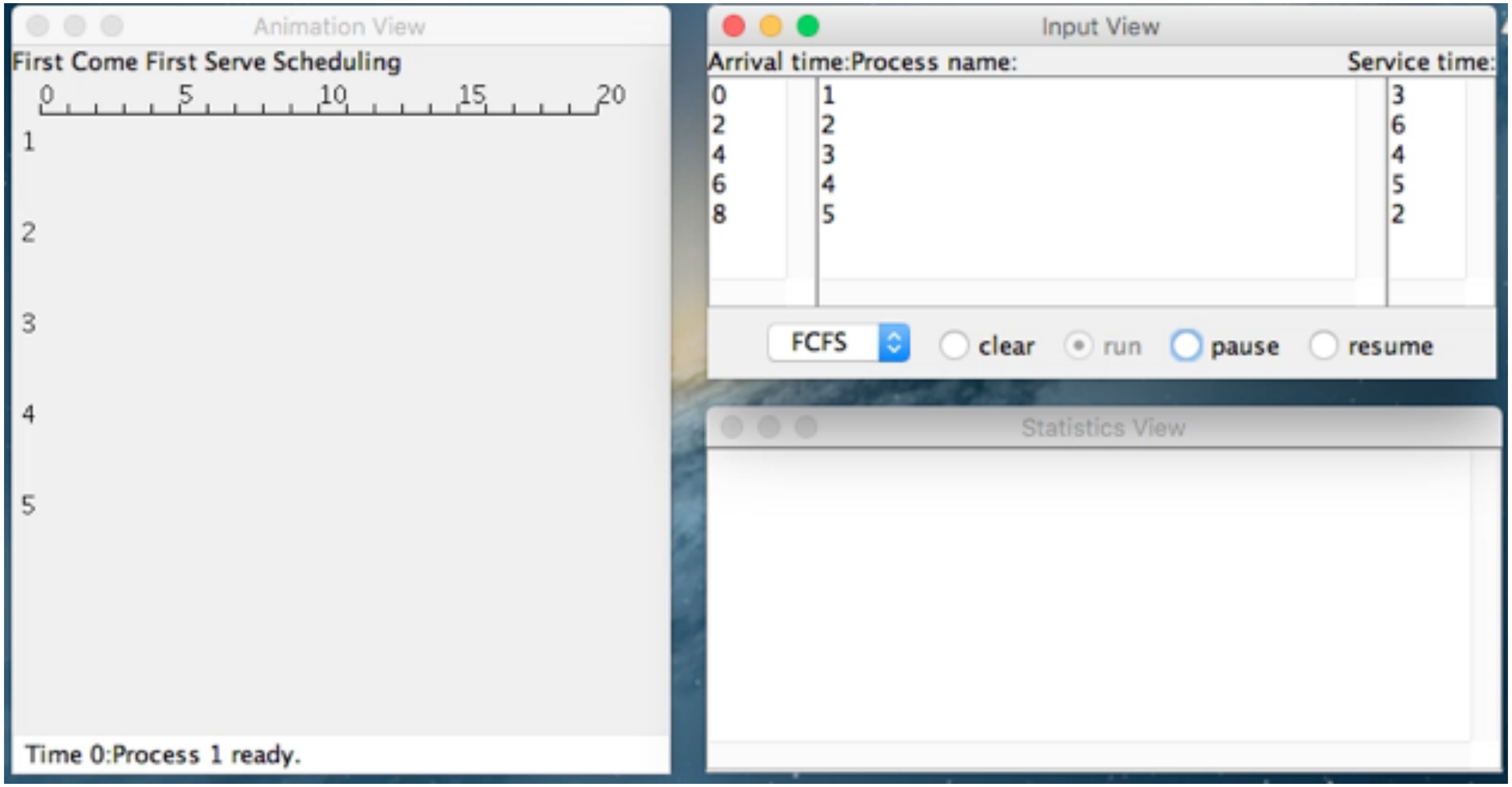
Scheduling in Batch Systems (1)

- First-Come First-Served
- Shortest Job First
- Shortest Remaining Time Next

Scheduling in Batch Systems (2)

- **First-Come First-Served:**
  - processes are assigned the CPU in the order they request it
  - when the jobs come in, they are put onto the end of the queue
  - when the running process blocks, the first process on the queue is run next
  - when a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue, behind all waiting processes

Scheduling in Batch Systems (3)



Arrival time:	Process name:	Service time:
0	1	3
2	2	6
4	3	4
6	4	5
8	5	2

Time 0:Process 1 ready.

First-Come First-Served Algorithm

Scheduling in Batch Systems (4)

- **Shortest Job First:**
  - several equally important jobs are sitting in the input queue waiting to be started
  - the scheduler picks the shortest job first, so the average turnaround time for each job would be minimal
  - suppose, there are four jobs A, B, C and D, with run times of 8, 4, 4 and 4 minutes. If a scheduler ran them in this order, average turnaround time would be 14 minutes ((8 + 12 + 16 + 20) / 4)
  - if a scheduler ran B, C, D and A, average turnaround time would be 11 minutes ((4 + 8 + 12 + 20) / 4)

Scheduling in Batch Systems (5)

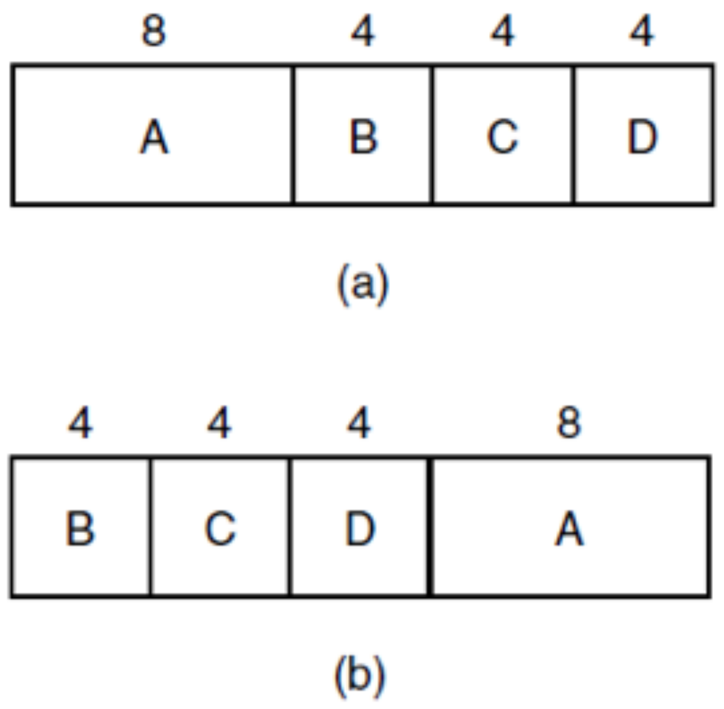
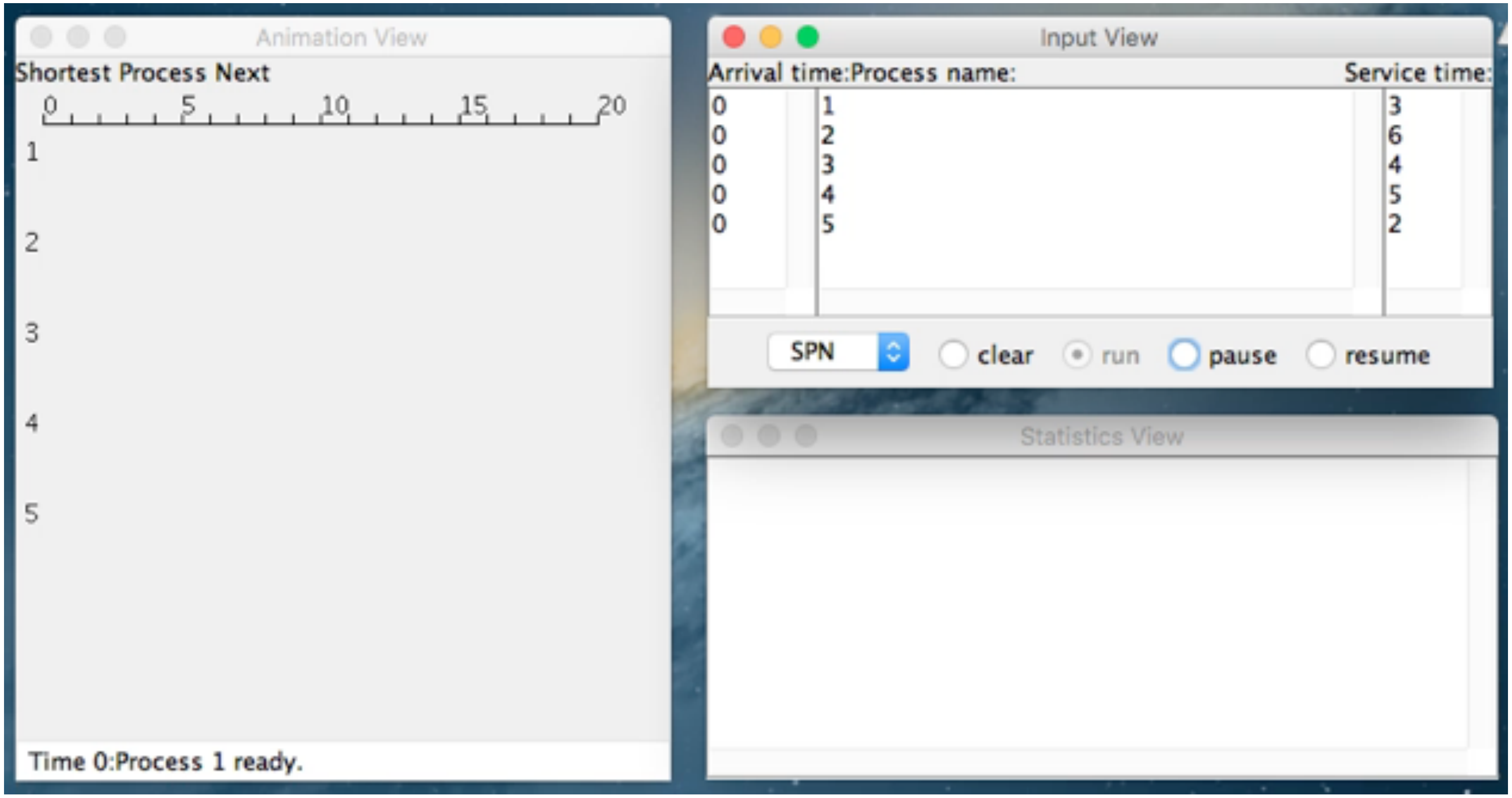


Figure 2-41. An example of shortest job first scheduling.
(a) Running four jobs in the original order. (b) Running them in shortest job first order.

Scheduling in Batch Systems (6)



Shortest Process Next

Time 0:Process 1 ready.

Arrival time:	Process name:	Service time:
0	1	3
0	2	6
0	3	4
0	4	5
0	5	2

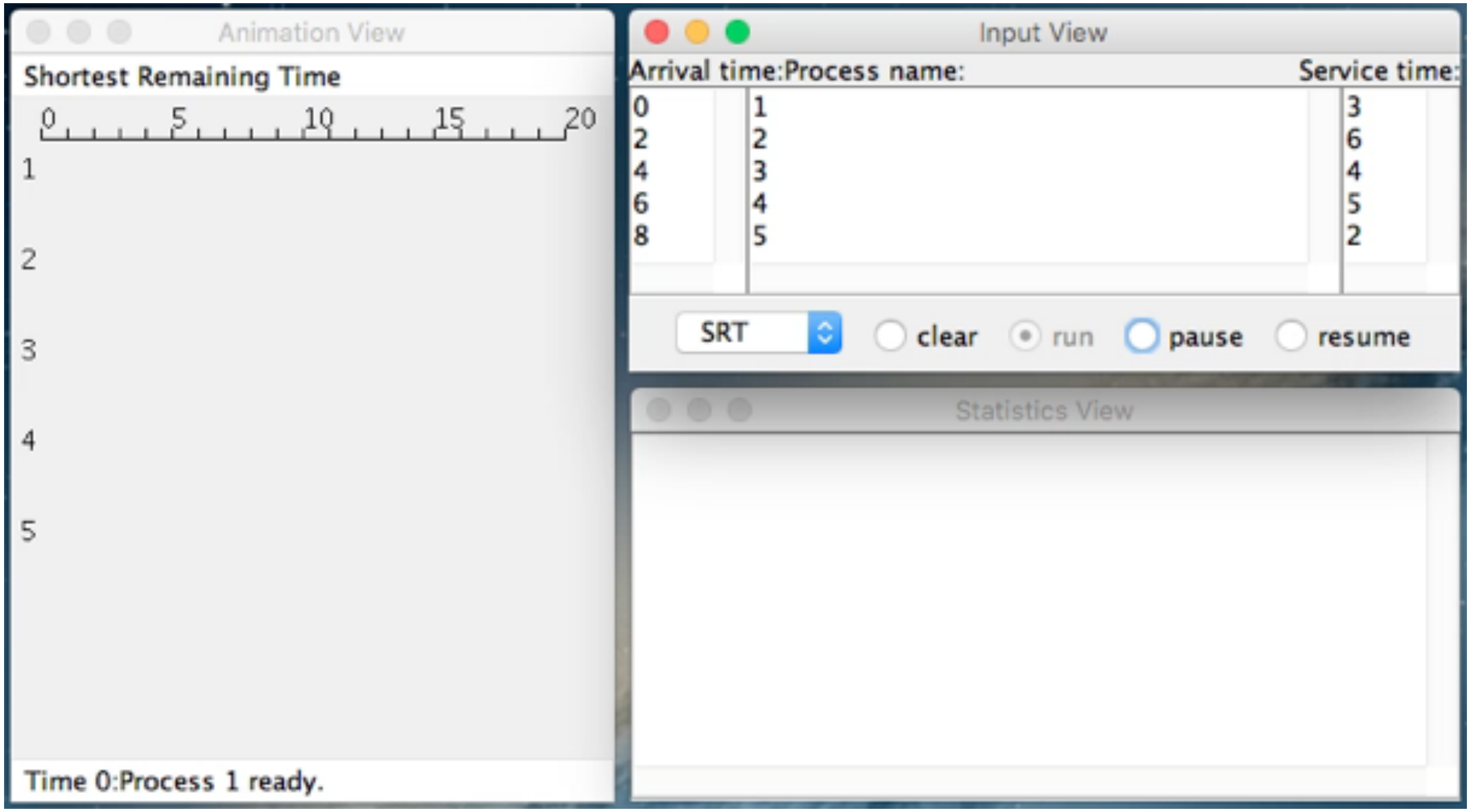
SPN   clear   [✓ run]   pause   resume

Shortest Job First Algorithm

Scheduling in Batch Systems (7)

- **Shortest Remaining Time Next:**
  - the scheduler always chooses the process whose remaining run time is the shortest
  - the run time has to be known in advance
  - when a new job arrives, its total time is compared to the current process' remaining time
  - if the new job needs less time to finish than the current process, the current process is suspended and the new job started

Scheduling in Batch Systems (5)



Time 0:Process 1 ready.

Arrival time:	Process name:	Service time:
0	1	3
2	2	6
4	3	4
6	4	5
8	5	2

Shortest Remaining Time First Algorithm

Scheduling in Interactive Systems (1)

- Round-Robin Scheduling
- Priority Scheduling
- Multiple Queues
- Shortest Process Next
- Guaranteed Scheduling
- Lottery Scheduling
- Fair-Share Scheduling

Scheduling in Interactive Systems (2)

- **Round-Robin Scheduling:**
  - each process is assigned a time interval, called a **quantum**
  - if the process is still running at the end of the quantum, the CPU is preempted and given to another process (Fig. 2-42)
  - if the process has blocked or finished before the quantum has elapsed, the CPU switching is done when the process blocks

Scheduling in Interactive Systems (3)

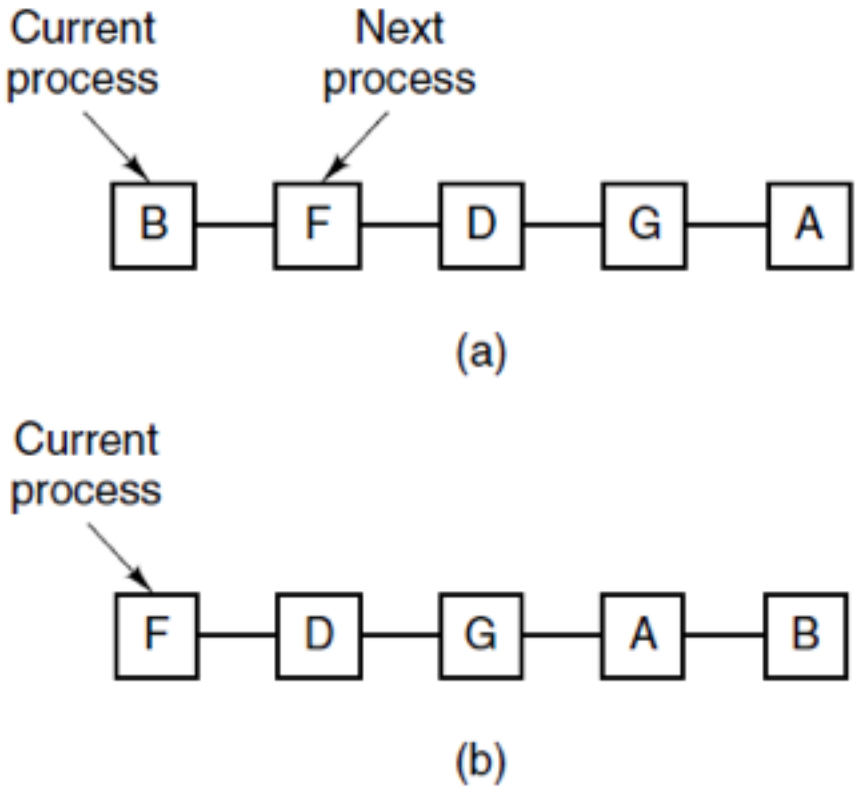
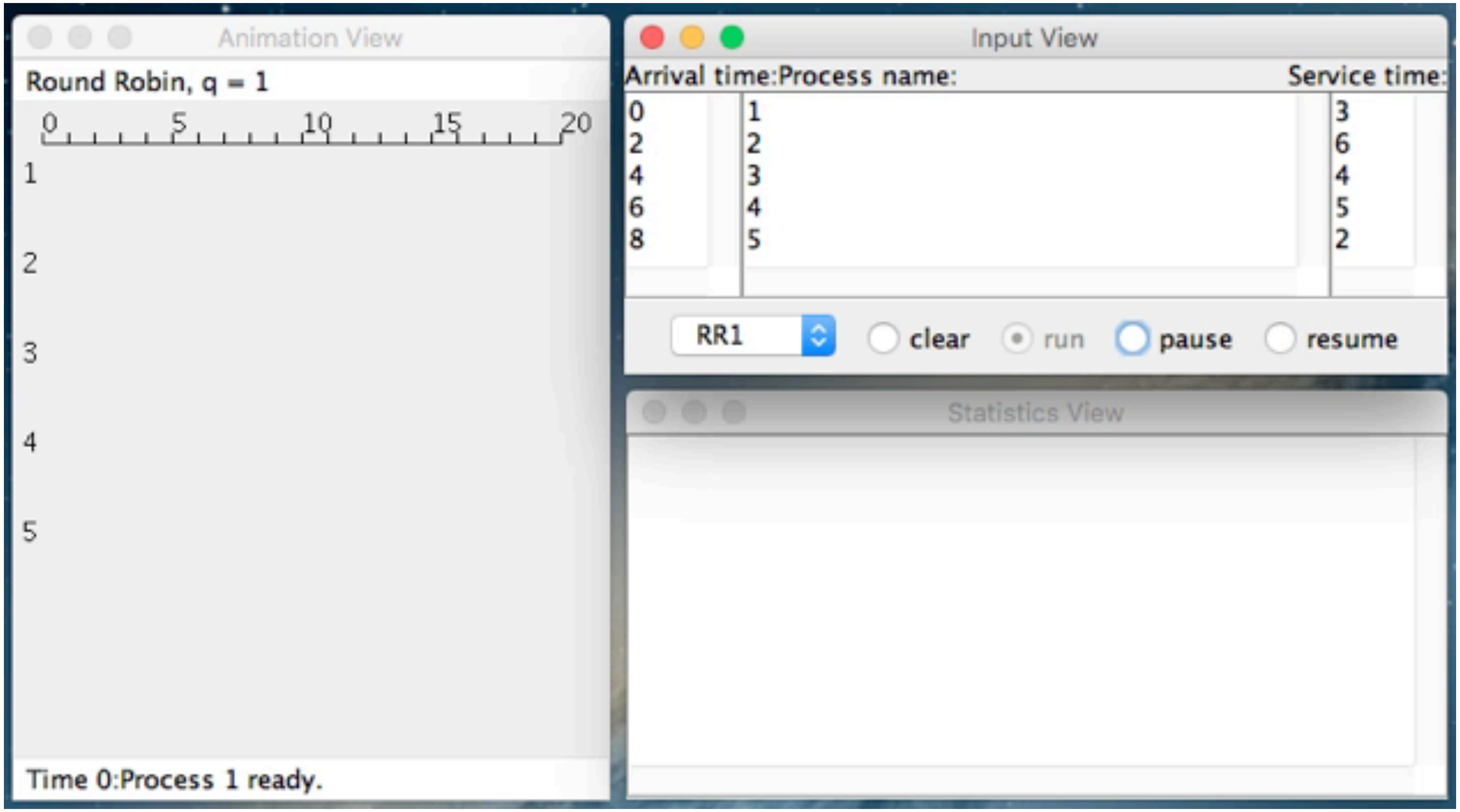


Figure 2-42. Round-robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Scheduling in Interactive Systems (4)



Round Robin, q = 1

Arrival time:	Process name:	Service time:
0	1	3
2	2	6
4	3	4
6	4	5
8	5	2

RR1 [✓ run]

Time 0:Process 1 ready.

Round-robin scheduling

Scheduling in Interactive Systems (5)

- **Priority Scheduling:**
  - each process is assigned a priority, and the runnable process with the highest priority is allowed to run
  - for example, a daemon process sending an email in the background should be assigned a lower priority than a process displaying a video film on the screen in real time
  - to prevent high-priority processes from running indefinitely, the scheduler may decrease the priority of the currently running process at each clock tick
  - alternatively, each process may be assigned a maximum time quantum that it is allowed to run

Scheduling in Interactive Systems (6)

- **Multiple Queues:**
  - it is more efficient to give CPU-bound processes a large quantum once in a while, rather than giving them small quanta frequently (to reduce swapping)
  - on the other hand, giving all processes a large quantum would mean poor response time
  - the solution is to set up priority classes
  - processes in the highest class run for one quantum; processes in the next-highest class run for two quanta; processes in the next one were run for four quanta, etc.
  - whenever a process uses up all the quanta allocated to it, it is moved down one class

Scheduling in Interactive Systems (6)

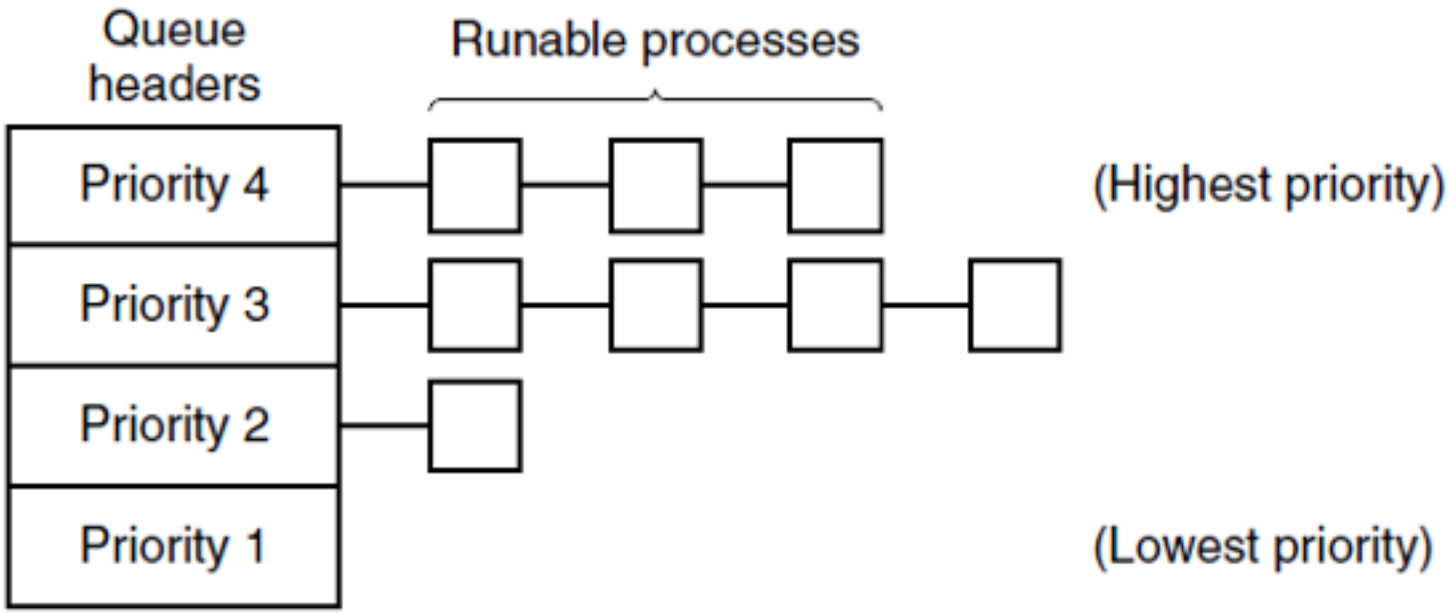
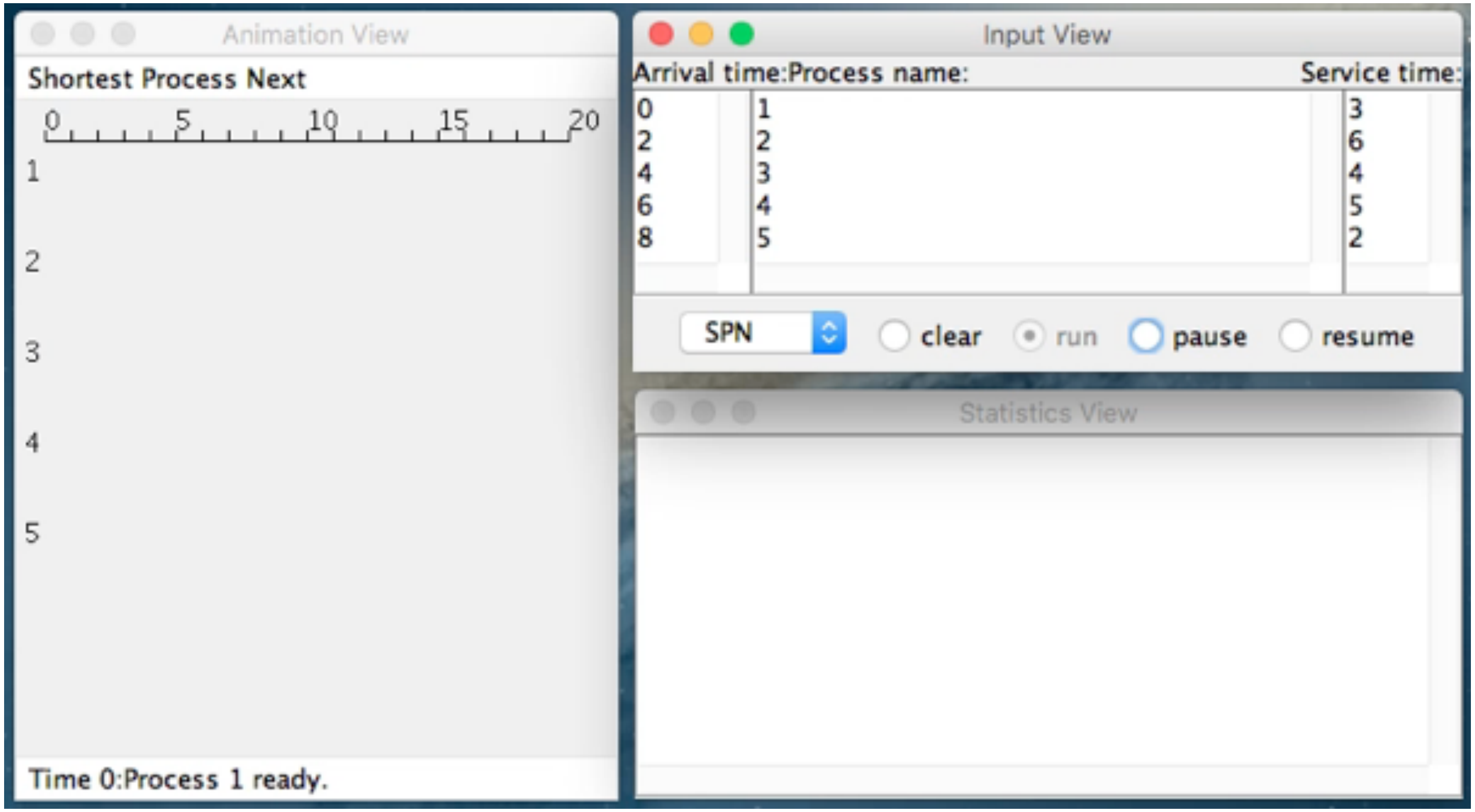


Figure 2-43. A scheduling algorithm with four priority classes.

Scheduling in Interactive Systems (7)

- Shortest Process Next:
  - analogue of the Shortest Job First algorithm in batch processing systems
  - interactive processes generally follow the pat- tern of wait for command, execute command, wait for command, execute command, etc.
  - if we regard the execution of each command as a separate "job," then we can minimize overall response time by running the shortest one first
  - the only problem is to figure out which of the currently runnable processes is the shortest one

Scheduling in Interactive Systems (8)



Arrival time:	Process name:	Service time:
0	1	3
2	2	6
4	3	4
6	4	5
8	5	2

Time 0:Process 1 ready.

Shortest Process Next Algorithm

Scheduling in Interactive Systems (9)

- **Guaranteed Scheduling:**
  - to make real promises to the users about performance and then live up to those promises
  - for example, if *n* users are logged in while you are working, you will receive about 1/*n* of the CPU power
  - on a single-user system with *n* processes running, all things being equal, each one should get 1/*n* of the CPU cycles

Scheduling in Interactive Systems (10)

- **Lottery Scheduling:**
  - the basic idea is to give processes lottery tickets for various system resources, such as CPU time
  - whenever a scheduling decision has to be made, a lottery ticket is chosen at random, and the process holding that ticket gets the resource
  - when applied to CPU scheduling, the system might hold a lottery 50 times a second, with each winner getting 20 msec of CPU time as a prize
  - more important processes can be given extra tickets, to increase their odds of winning

Scheduling in Interactive Systems (11)

- **Fair-Share Scheduling:**
  - a scheduling algorithm which takes into consideration the owners of the processes and the number of processes each user owns
  - if two users have each been promised 50% of the CPU, they will each get that, no matter how many processes they have in existence
  - suppose, there are two users – the first one owns the processes A, B, C and D; the second one owns only one process E. If we have used round-robin scheduling, the process scheduling sequence would be ABCDEABCDE…
  - in a system with fair-share scheduling, however, it will be AEBECEDEAEBECEDE…

Scheduling in Real-Time Systems (1)

- Time plays an essential role
- For example, the computer in a CD player gets the bits as they come off the drive and must convert them into music within a very tight time interval
- Other examples are:
  - patient monitoring in a hospital intensive-care unit
  - the autopilot in an aircraft
  - robot control in an automated factory

Scheduling in Real-Time Systems (2)

- Categories of the real-time systems:
  - Hard real time — there are absolute deadlines that must be met
  - Soft real time — missing an occasional deadline is undesirable, but nevertheless tolerable
- Categories of events:
  - Periodic — occur at regular intervals
  - Aperiodic — occur unpredictably

Scheduling in Real-Time Systems (3)

- If there are *m* periodic events and event *i* occurs with period P_i and requires C_i seconds of CPU time to handle each event, then the load can be handled only if

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

- A real-time system that meets this criterion is said to be **schedulable**

Thread Scheduling (1)

- When several processes each have multiple threads, there are two levels of parallelism present: processes and threads
- Scheduling in such systems differs depending on whether user-level threads or kernel-level threads (or both) are supported

Thread Scheduling (2)

- **User-level threads:**
  - Since the kernel is not aware of the existence of threads, it picks a process and gives it control for its quantum
  - The thread scheduler inside the process decides which thread to run
  - One thread might consume all of the process's time until it is finished, however, it will not affect the other processes
  - The only constraint is the absence of a clock to interrupt a thread that has run too long. Since threads cooperate, this is usually not an issue

Thread Scheduling (3)

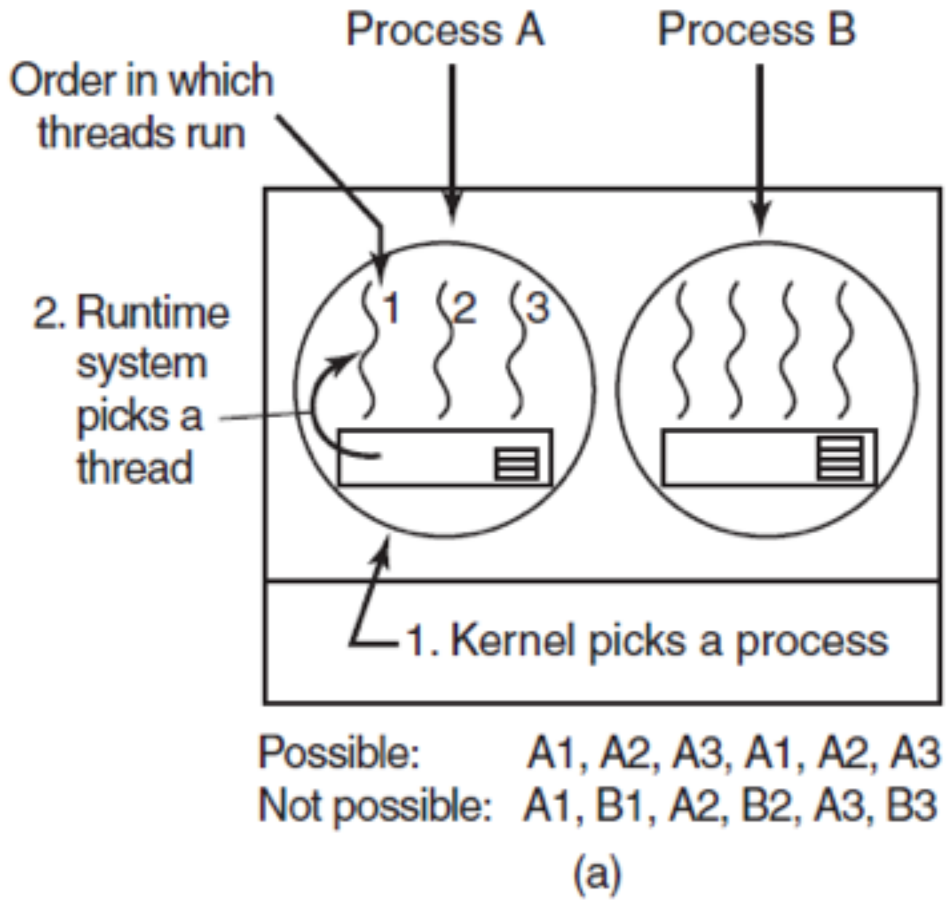

Process A
Process B
Order in which threads run
1
2
3
2. Runtime system picks a thread
1. Kernel picks a process
Possible: A1, A2, A3, A1, A2, A3
Not possible: A1, B1, A2, B2, A3, B3
(a)

Figure 2-44. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Thread Scheduling (4)

- Kernel-level threads:
  - The kernel picks a particular thread to run. It does not have to take into account which process the thread belongs to, but it can if it wants to
  - The thread is given a quantum and is for-cibly suspended if it exceeds the quantum

Thread Scheduling (5)

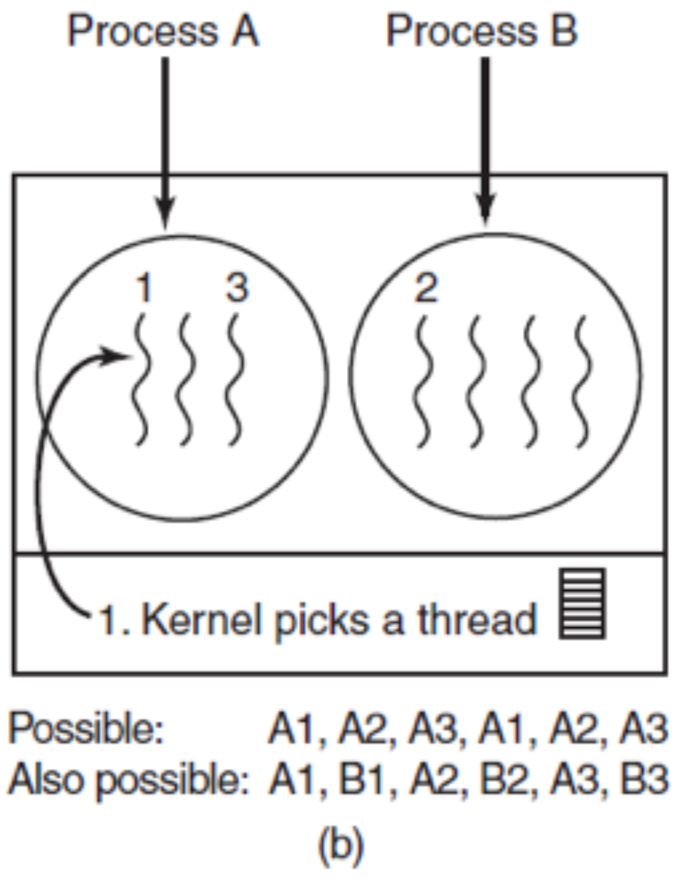

Process A
Process B
1
3
2
1. Kernel picks a thread
Possible: A1, A2, A3, A1, A2, A3
Also possible: A1, B1, A2, B2, A3, B3
(b)

Figure 2-44. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

Thread Scheduling (6)

- Major differences:
  - **Performance**. Doing a thread switch with user-level threads takes a handful of ma-chine instructions. With kernel-level threads it requires a full context switch
  - **Thread scheduling**. User-level threads can employ an application-specific thread scheduler that can tune an application better than the kernel can

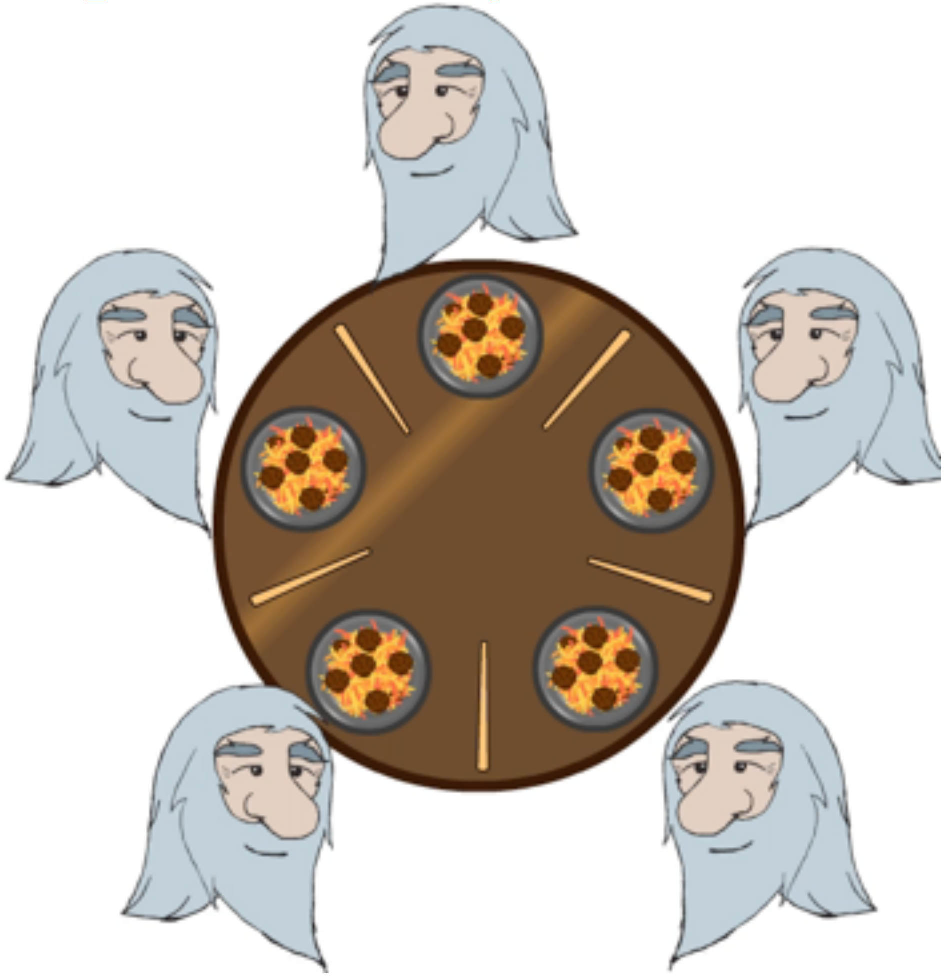
The Dining Philosophers Problem (1)

- Five philosophers are sitting around a circular table eating spaghetti but require two forks to do so. There is one fork between each philosopher (for a total of five forks). The method the philosophers use to eat is this:
    1. Philosophize for a while.
    2. Pick up the fork on your left.
    3. Wait until the fork on your right is available.
    4. Pick up the fork on your right.
    5. Eat.
    6. Put the forks down.
    7. Go back to step 1.

The Dining Philosophers Problem (2)

- Sooner or later everyone at the table will end up with the fork on their left in their hand and waiting for the fork on their right. But because everyone is holding on to the fork their neighbor is waiting for and won't put it down until they've eaten, the philosophers are in a **deadlock** state

The Dining Philosophers Problem (3)



Lunch time in the Philosophy Department

The Dining Philosophers Problem (4)

```
#define N 5                              /* number of philosophers */

void philosopher(int i)                  /* i: philosopher number, from 0 to 4 */
{
        while (TRUE) {
                think( );                /* philosopher is thinking */
                take_fork(i);            /* take left fork */
                take_fork((i+1) % N);    /* take right fork; % is modulo operator */
                eat( );                  /* yum-yum, spaghetti */
                put_fork(i);             /* put left fork back on the table */
                put_fork((i+1) % N);     /* put right fork back on the table */
        }
}
```

Figure 2-46. A nonsolution to the dining philosophers problem.

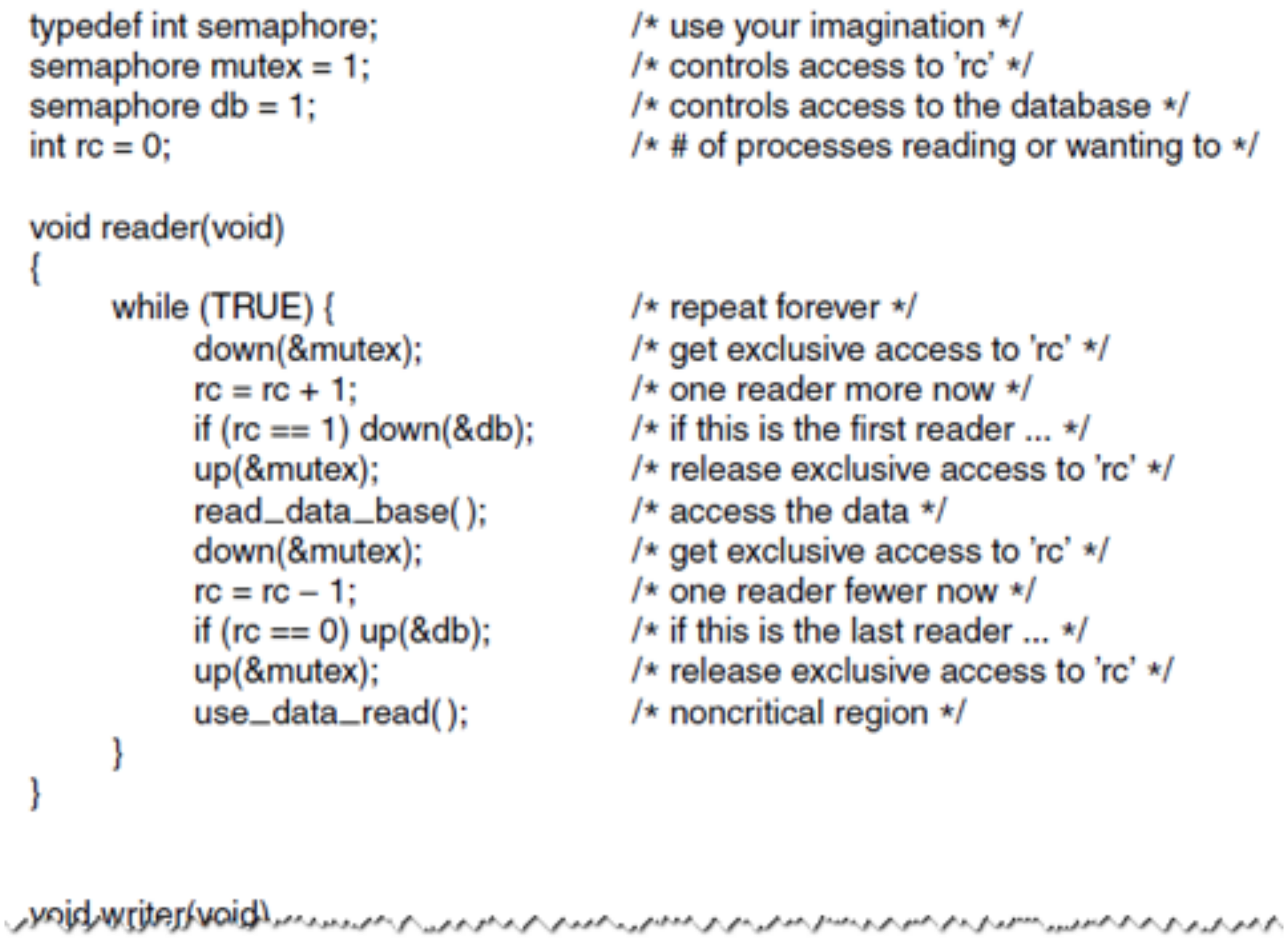
The Dining Philosophers Problem (5)

- The algorithm could be modified so that after taking the left fork, the philosopher checks to see if the right fork is available. If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process
- All the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever
- A situation like this, in which all the programs continue to run indefinitely but fail to make any progress, is called **starvation**

The Readers and Writers Problem (1)

- Imagine, for example, an airline reservation system, with many competing processes wishing to read and write it
- It is acceptable to have multiple processes reading the database at the same time, but if one process is updating (writing) the database, no other processes may have access to the database, not even readers
- The question is how do you program the readers and the writers?

The Readers and Writers Problem (2)



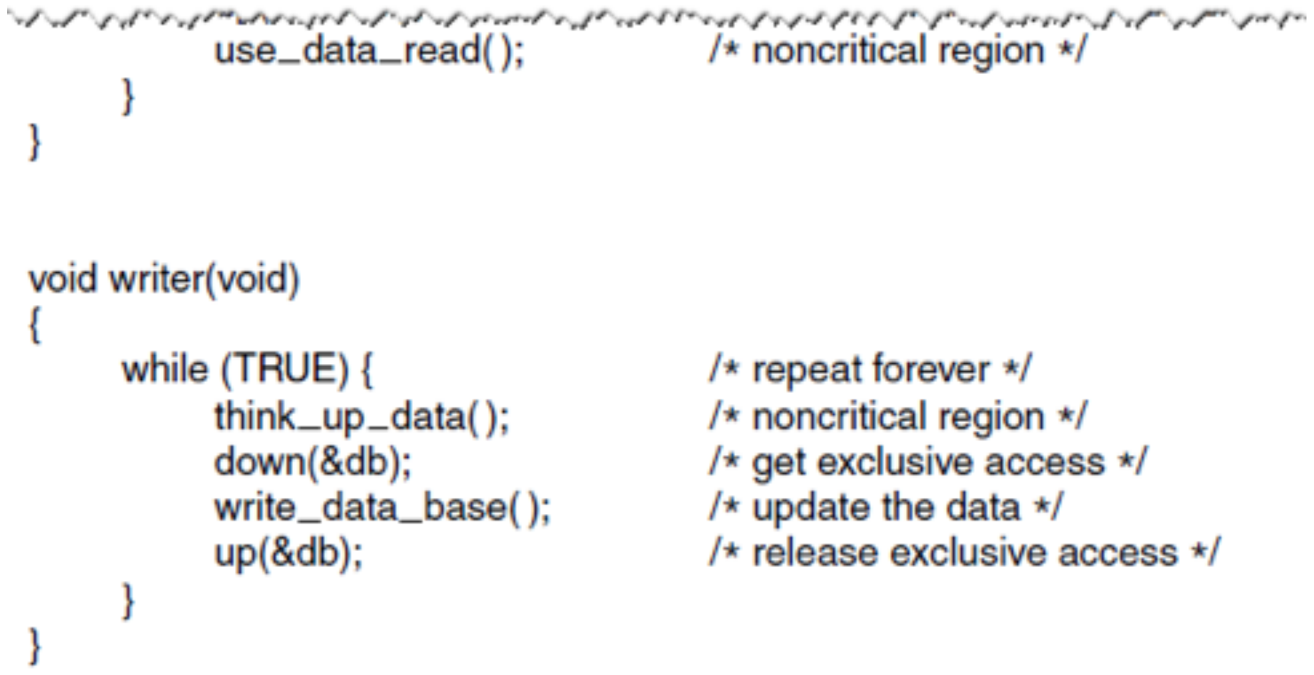
```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
     while (TRUE) {                     /* repeat forever */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc + 1;                  /* one reader more now */
          if (rc == 1) down(&db);       /* if this is the first reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          read_data_base();             /* access the data */
          down(&mutex);                 /* get exclusive access to 'rc' */
          rc = rc – 1;                  /* one reader fewer now */
          if (rc == 0) up(&db);         /* if this is the last reader ... */
          up(&mutex);                   /* release exclusive access to 'rc' */
          use_data_read();              /* noncritical region */
     }
}

void writer(void)
```

Figure 2-48. A solution to the readers and writers problem.

The Readers and Writers Problem (3)



```
            use_data_read();            /* noncritical region */
        }
}


void writer(void)
{
        while (TRUE) {                  /* repeat forever */
            think_up_data();            /* noncritical region */
            down(&db);                  /* get exclusive access */
            write_data_base();          /* update the data */
            up(&db);                    /* release exclusive access */
        }
}
```

Figure 2-48. A solution to the readers and writers problem.

End

Week 05 – Lecture 2

References

- Tanenbaum & Bo, Modern Operating Systems: 4th edition, 2013
  Prentice-Hall, Inc.
- http://rodrev.com/graphical/programming/Deadlock.swf
- http://www.utdallas.edu/~ilyen/animation/cpu/program/prog.html
- https://courses.cs.vt.edu/csonline/OS/Lessons/Processes/index.html
- http://inventwithpython.com/blog/2013/04/22/multithreaded-python-tutorial-with-threadworms/