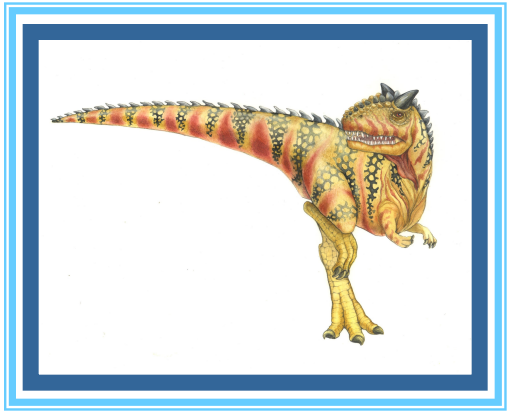
Spring 2022 COMP 3511
Review #3



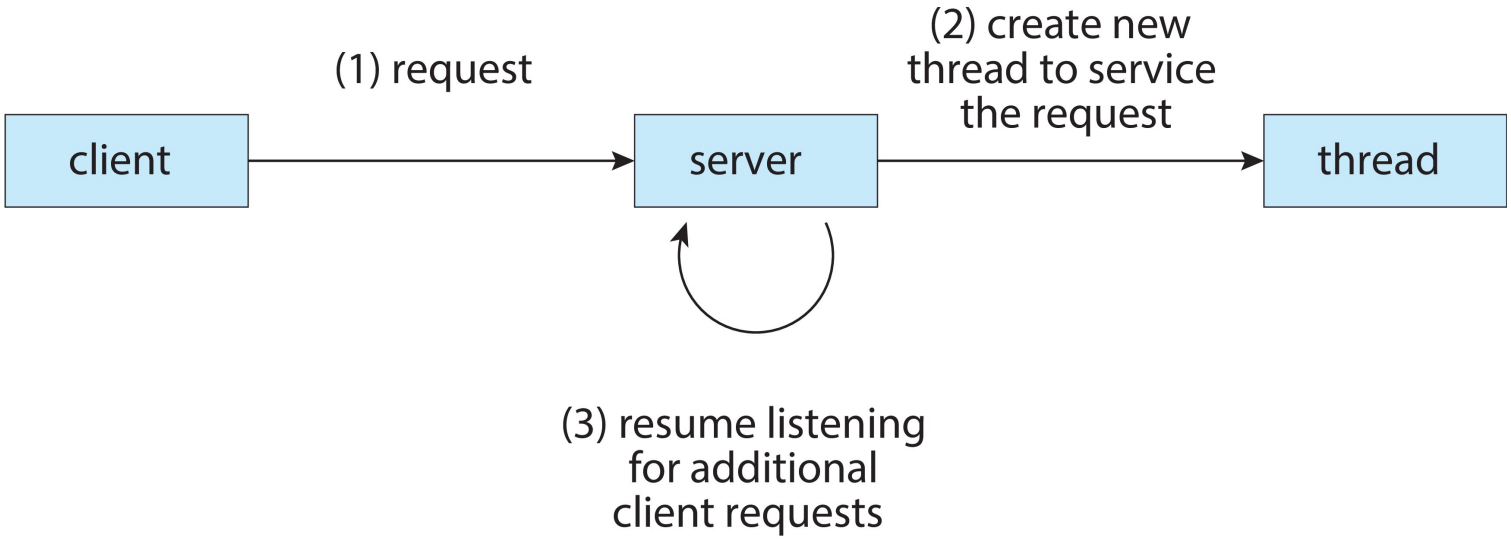
Coverages

- Thread
- CPU Scheduling

Asynchronous Threading

Asynchronous threading

- When a parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another. Because the threads are independent, there is typically little data sharing between them



Synchronous Threading

Synchronous threading

- Synchronous threading occurs when a parent thread creates one or more children and then must wait for all of its children threads to terminate before it continues execution. Thus, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed. Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined, can the parent resume execution.
- Typically, synchronous threading involves significant data sharing among threads
- For example, this can be used when the parent thread may combine the results calculated by its many children threads

Pthreads

- **Pthreads** - POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a ***specification*** for thread behaviour, not an ***implementation***. Different Unix-like OS implements this differently
- The C program shown next demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.
- In this Pthreads program, separate threads begin execution in a specified function - the **`runner()`** function
  - When this program begins, a single thread of control (the parent thread) begins in **`main()`**. After some initialization, **`main()`** creates a second thread (the child thread) running **`runner()`** function. Both threads share the global data **`sum`**
  - This program follows the thread *create/join* strategy, whereby after creating summation thread – the parent thread waits for the completion of child thread
  - Once the summation thread has returned, the parent thread will output the value of the shared data **`sum`**

Pthreads

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

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Pthreads Example

- In a Pthreads program, separate threads begin execution, specifically
  - All Pthreads programs must include the **`pthread.h`** header file
  - The **`pthread_t`** declares the identifier for the thread to be created
  - The **`pthread_attr_t`** represents the attributes for the thread (stack size and scheduling information. Here default attributes are used)
  - A thread created with **`pthread_create()`**, with the thread identifier, attributes, function (i.e., the **`runner()`**), the integer parameter (i.e., **`argv[1]`**)
  - The program has two threads: the initial (or parent) thread in **`main()`** and the summation (or child) thread performing summation in the **`runner()`** function
  - The parent thread waits for its child to terminate by calling **`pthread_join()`** function
  - The summation thread will terminate when it calls function **`pthread_exit()`**

pthread_create function

- `#include <pthread.h>`
- ```
  int pthread_create( pthread_t * thread,
      const pthread_attr_t * attr,
      void * (*start_routine)(void*),
      void * arg);
  ```
  - thread, is a pointer to a structure of type pthread_t;
  - attr, is used to specify any attributes this thread might have;
  - Function pointer;
  - arg, the argument to be passed to the function where the thread begins execution.


  ```

Pthreads Example: Create & Join

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

void *thread_function(void *arg) {
  for (int i=0; i<8; i++) {
    printf("Thread working...! %d \n",i);
  }
  return NULL;
}

int main(void) {
  pthread_t mythread;
  if (pthread_create(&mythread, NULL, thread_function, NULL)) {
    printf("error creating thread.");
    abort();
  }
  if (pthread_join (mythread, NULL)) {
    printf("error join thread.");
    abort();
  }
  printf("thread done! \n");
  return 0;
}
```

Pthreads Example (Cont.)

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example (Cont.)

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Test the pthread program

- Compile the C program.
- pthread is required in the linking process to generate the executable

```
> gcc pthread_demo.c -lpthread -o pthread_demo
```

```
> ./pthread_demo 2
sum of 1..2 = 3
```

```
> ./pthread_demo 1000
sum of 1..1000 = 500500
```

Pthreads Code for Joining 10 Threads

- A simple method for waiting on multiple threads using **`pthread_join()`** function is to enclose the operation within a simple for loop.
- For example, join on ten threads using the Pthread code below

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Another Pthreads example

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Trace the threads(1)

main	Thread 1	Thread 2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Trace the threads(2)

main	Thread 1	Thread 2
starts running		
prints "main: begin"		
creates Thread 1		
	runs	
	prints "A"	
	returns	
creates Thread 2		
		runs
		prints "B"
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints "main: end"		

Time to consider…

Will there be any other possible outputs?

Will Thread1 runs after Thread2?

Trace the threads(3)

main	Thread 1	Thread 2
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Extend this example…

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

#define LENGTH 2000

void *mythread(void *arg) {
    printf("begin of %s\n", (char *) arg);
    for (int i = 0; i < LENGTH; ++i)
    {
        int things_in_thread = i + 3;
    }
    printf("end of %s\n", (char *) arg);
    return NULL;
}
int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    printf("main: start\n");
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: finish\n");
    return 0;
}
```

What is the possible outputs?

How many possible outputs of this example?

Considering different orders, there should be 6 possible outputs

```
qyx@qyxdeMacBook-Pro Pthread_Examples % ./a.out
main: start
begin of A
begin of B
end of B
end of A
main: finish
```

```
qyx@qyxdeMacBook-Pro Pthread_Examples % ./a.out
main: start
begin of B
begin of A
end of B
end of A
```

```
qyx@qyxdeMacBook-Pro Pthread_Examples % ./a.out
main: start
begin of B
begin of A
end of A
end of B
main: finish
```

```
qyx@qyxdeMacBook-Pro Pthread_Examples % ./a.out
main: start
begin of B
end of B
begin of A
end of A
main: finish
```

```
qyx@qyxdeMacBook-Pro Pthread_Examples % ./a.out
main: start
begin of A
end of A
begin of B
end of B
main: finish
```

```
qyx@qyxdeMacBook-Pro Pthread_Examples % ./a.out
main: start
begin of A
begin of B
end of A
end of B
main: finish
```

Clone

- The **`clone()`** system call creates a new child process or called a task.
- Compared to **`fork()`**, **`clone()`** provides more precise control over what pieces of execution context are shared between the calling parent process and the child process – for example, **`clone()`** can control whether the two processes share the virtual address space, the table of file descriptors, and the table of signal handlers.

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.

Simple example of clone()

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <sched.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int fn(void *arg)
{
   printf("\nINFO: This code is running under child process.\n");

   int i = 0;

   int n = atoi(arg);

   for ( i = 1 ; i <= 10 ; i++ )
      printf("%d * %d = %d\n", n, i, (n*i));

   printf("\n");

   return 0;
}

int main(int argc, char *argv[])
{
   printf("Hello, World!\n");

   void *pchild_stack = malloc(1024 * 1024);
   if ( pchild_stack == NULL ) {
      printf("ERROR: Unable to allocate memory.\n");
      exit(EXIT_FAILURE);
   }

   int pid = clone(fn, pchild_stack + (1024 * 1024), SIGCHLD, argv[1]);
   if ( pid < 0 ) {
        printf("ERROR: Unable to create the child process.\n");
        exit(EXIT_FAILURE);
   }

   wait(NULL);

   free(pchild_stack);

   printf("INFO: Child process terminated.\n");
   return 0;
}
```

Outputs of this example

```
$ ./simple_example_of_clone 7
Hello, World!

INFO: This code is running under child process.
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70

INFO: Child process terminated.
```

Example of clone()

```c
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#define STACK_SIZE 65536
char *heap;
static int child_func(void *arg) {
    char *buf = (char *)arg;
    printf("Child sees buf = %s\n", buf);
    printf("Child sees heap = %s\n", heap);
    strcpy(buf, "Hello from child");
    strcpy(heap, "Bye");
    return 0;
}
int main(int argc, char *argv[]) {
    unsigned long flags = 0;
    char buf[256];
    char *stack = malloc(STACK_SIZE);
    int status;
    heap = malloc(1024);
    if (argc == 2 && !strcmp(argv[1], "vm")) flags |= CLONE_VM;
    strcpy(buf, "Hello from parent");
    strcpy(heap, "Hey");
    clone(child_func, stack + STACK_SIZE, flags | SIGCHLD, buf);
    wait(&status);
    printf("Child exited with status: %d\n", status);
    printf("buf = %s\n", buf);
    printf("heap = %s\n", heap);
    return 0;
}
```

Example of clone()

- If we pass CLONE_VM flag to **`clone()`**, the memory space will be shared between the calling process and the child process, so the modifications made on the child process will be reflected in the calling process:

```
$ gcc clone_vm.c -o clone
$ ./clone
Child sees buf = Hello from parent
Child sees heap = Hey
Child exited with status: 0
buf = Hello from parent
heap = Hey
$ ./clone vm
Child sees buf = Hello from parent
Child sees heap = Hey
Child exited with status: 0
buf = Hello from child
heap = Bye
```

No CLONE_VM flag, buf and heap remains unchanged

Pass CLONE_VM flag, buf and heap changed

CPU Scheduler

- The CPU scheduling is the task of selecting a waiting process from the ready queue (one or multiple queues) and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- The non-preemptive scheduling is invoked when the process running on the CPU gives up the CPU voluntarily
  - Switches from running to waiting state, e.g., I/O request, or wait()
  - Terminates
- The preemptive scheduling - where the CPU can be taken away from a process involuntarily can further be invoked when
  - Switches from running to ready state, e.g., quantum (timer) expires in RR scheduling
  - Switches from waiting to ready (e.g., completion of I/O) or from new to ready (new process with higher priority) - in SRTF or MLFQ scheduling

Scheduling Criteria

- Waiting time – amount of time a process waiting in the ready queue
- Turnaround time – defined as the time at which the process completes minus the time at which the process arrived in the system
  - Considering single CPU burst, turnaround time = waiting time + CPU burst time
- Response time – the amount of time it takes from when a request was submitted until the first response is produced
  - This is relevant to interactive programs (typically using RR scheduling)
- Fairness
  - Resources such as CPU are utilized in some “fair” manner
- Performance (such as turn-around time) and fairness are often at odds, in which optimizing performance often sacrifice fairness

Exponential Averaging Algorithm

- This is commonly used to estimate a future value based on historical data – the recent historical value(s) weight more than the past historical value(s)
  1. t_n = actual length of nth CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

If we expand the formula, we get:

$$\begin{aligned}
\tau_{n+1} &= \alpha t_n + (1-\alpha)\tau_n \\
&= \alpha t_n + (1-\alpha) \cdot (\alpha t_{n-1} + (1-\alpha)\tau_{n-1}) \\
&= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \tau_{n-1} \\
&= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \cdot (\alpha t_{n-2} + (1-\alpha)\tau_{n-2}) \\
&= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + (1-\alpha)^3 \tau_{n-2} \\
&= \cdots \\
&= \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1} \tau_0
\end{aligned}$$

Exponential Averaging Algorithm

- This is commonly used to estimate a future value based on historical data – the recent historical value(s) weight more than the past historical value(s)
  1. t_n = actual length of nth CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

If we expand the formula, we get:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$$

- All past values (ie the length of the CPU burst) matter, since they ($t_n, t_{n-1}, t_{n-2}, \ldots$) are included in the formula
- But recent values matter more. The past value effect is diminished exponentially fast, with each time multiple by $(1-\alpha)$.
- For example, if $\alpha = 0.5$, the effect after 8 rounds, $(1-\alpha)^{9+1} = 1/1024$, which can be ignored.

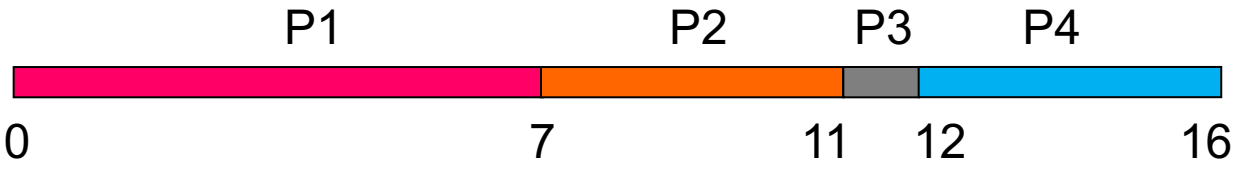
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served（FCFS）means the firstly arrived process can be firstly served.
- FCFS is also known as First-In, First-Out (FIFO).
- Although FCFS is simple, it also has many good properties, eg
  - easy to implement
  - works pretty well under specific assumptions

FCFS Scheduling: Example

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4



- We assume the processes arrive in this order: P1, P2, P3, P4
- Waiting time for P1=0, P2=7, P3=11, P4=12
- Average waiting time: (0+7+11+12)/4=7.5

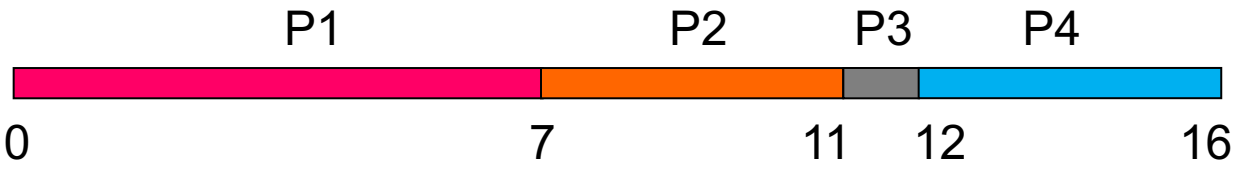
Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After quantum expires, the process is preempted (OS timer interrupt) and added to the end of the ready queue.
- Suppose n processes in the ready queue and the time quantum is q
  - In chunks of at most q time units at once.
  - No process waits more than (n-1)q time units
- Performance
  - q large $\Longrightarrow$ FCFS
  - q small $\Longrightarrow$ Interleaved (resemble process sharing)
  - q must be large with respect to context switch, otherwise overhead is too high

RR Scheduling: Example 1

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4

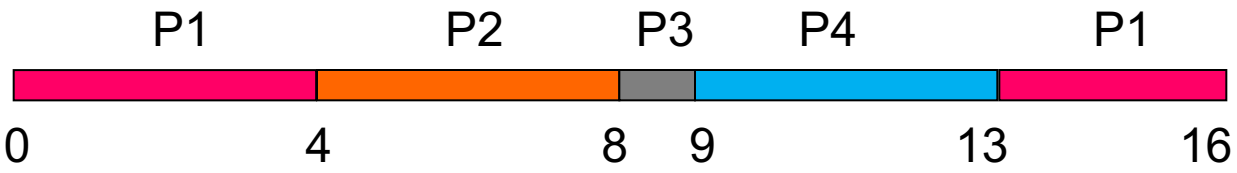


- When time quantum q is 8, which is larger than the longest burst time among the process.
  - The Gantt chart is the same as that of FCFS.
- Waiting time for P1=0, P2=7, P3=11, P4=12
- Average waiting time: (0+7+11+12)/4=7.5

RR Scheduling: Example 2

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4

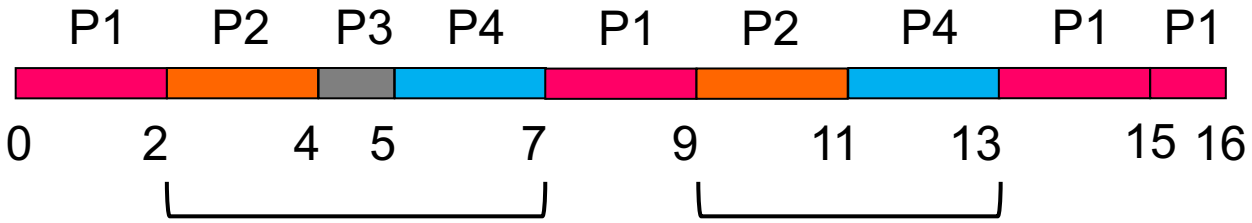


- When time quantum q is 4
- Waiting time for P1=(13-4)=9, P2=4, P3=8, P4=9
- Average waiting time: (9+4+8+9)/4=7.5

RR Scheduling: Example 3

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4

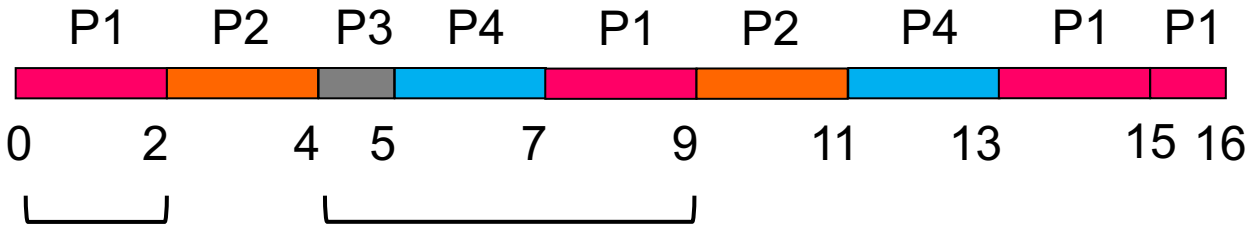


- When time quantum q is 2
- Waiting time for P1=5+4=9

RR Scheduling: Example 3

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4

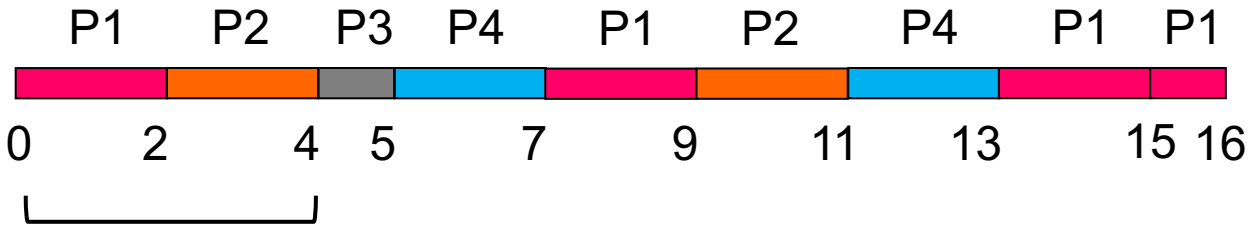


- When time quantum q is 2
- Waiting time for P1=9, P2=2+5=7

RR Scheduling: Example 3

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4

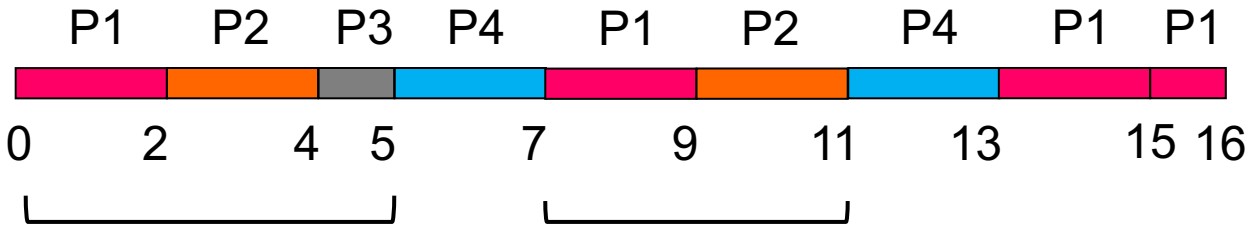


- When time quantum q is 2
- Waiting time for P1=9, P2=7, P3=4

RR Scheduling: Example 3

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4

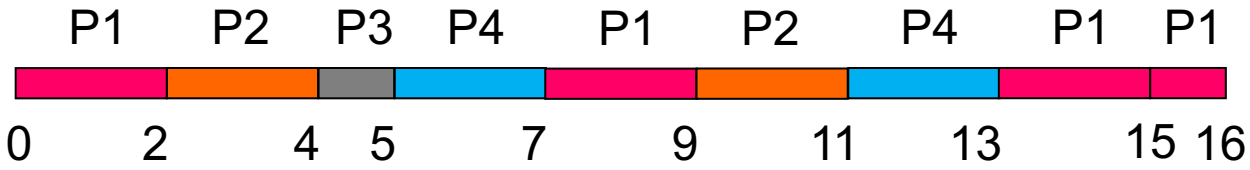


- When time quantum q is 2
- Waiting time for P1=9, P2=7, P3=4, P4=5+4=9

RR Scheduling: Example 3

Process	Burst Time
P1	7
P2	4
P3	1
P4	4

We assume the arrival order is P1, P2, P3, P4

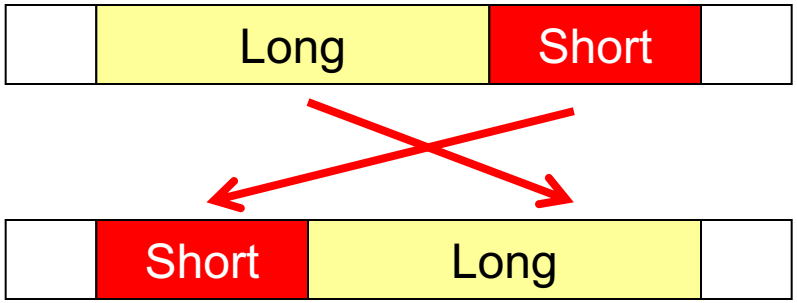


- When time quantum q is 2
- Waiting time for P1=9, P2=7, P3=4, P4=9
- Average waiting time: (9+7+4+9)/4=7.25 (better than FCFS, 7.5)

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

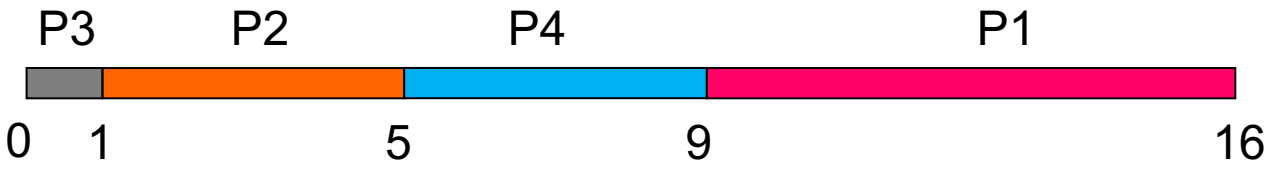
- Two schemes:
  - Non-preemptive – once the CPU is allocated to a process, it cannot be preempted until the completion of its current CPU burst



SJF is optimal – yields the minimum average waiting time for a given set of processes

SJF Scheduling: Example

Process	Burst Time	Arrival Time
P1	7	0
P2	4	0
P3	1	0
P4	4	0



- Since P2 and P4 have the same burst time, we first schedule P2, then P4
- Waiting time for P1=9, P2=1, P3=0, P4=5
- Average waiting time: (9+1+0+5)/4=3.75 (better than FCFS and RR)

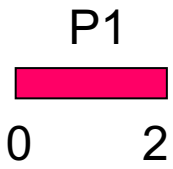
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time

- Two schemes:
  - Non-preemptive – once the CPU is allocated to a process, it cannot be preempted until the completion of its current CPU burst
  - Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is also know as the Shortest-Remaining-Time-First (SRTF)

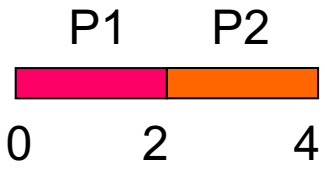
SRTF Scheduling: Example

Process	Burst Time	Arrival Time	Remaining Time
P1	7	0	7
P2	4	2	
P3	1	4	
P4	4	5	



SRTF Scheduling: Example

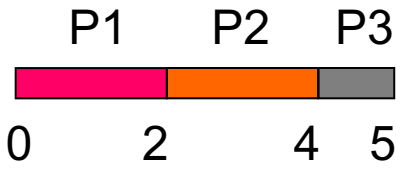
Process	Burst Time	Arrival Time	Remaining Time
P1	7	0	5
P2	4	2	4
P3	1	4	
P4	4	5	



- P1 gets preempted at time 2

SRTF Scheduling: Example

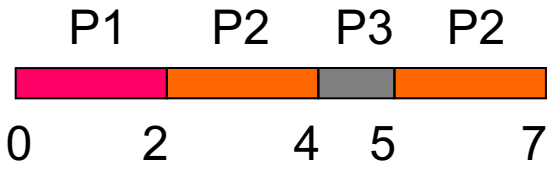
Process	Burst Time	Arrival Time	Remaining Time
P1	7	0	5
P2	4	2	2
P3	1	4	1
P4	4	5	



- P1 gets preempted at time 2
- P2 gets preempted at time 4

SRTF Scheduling: Example

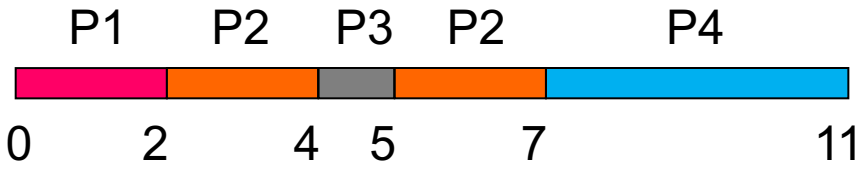
Process	Burst Time	Arrival Time	Remaining Time
P1	7	0	5
P2	4	2	2
P3	1	4	0
P4	4	5	4



- P1 gets preempted at time 2
- P2 gets preempted at time 4

SRTF Scheduling: Example

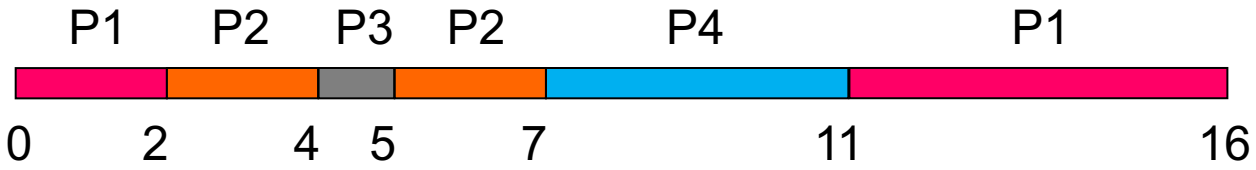
Process	Burst Time	Arrival Time	Remaining Time
P1	7	0	5
P2	4	2	0
P3	1	4	0
P4	4	5	4



- P1 gets preempted at time 2
- P2 gets preempted at time 4

SRTF Scheduling: Example

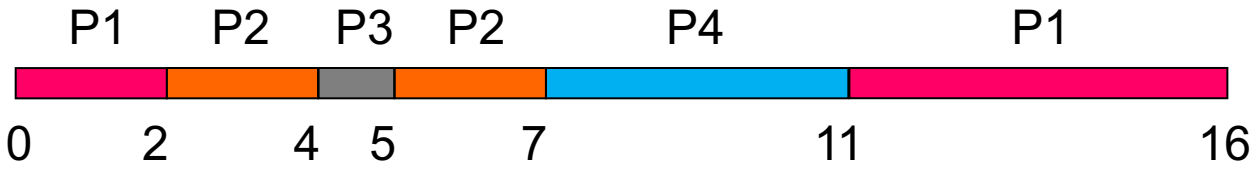
Process	Burst Time	Arrival Time	Remaining Time
P1	7	0	0
P2	4	2	0
P3	1	4	0
P4	4	5	0



- P1 gets preempted at time 2
- P2 gets preempted at time 4

SRTF Scheduling: Example

Process	Burst Time	Arrival Time	Remaining Time
P1	7	0	0
P2	4	2	0
P3	1	4	0
P4	4	5	0



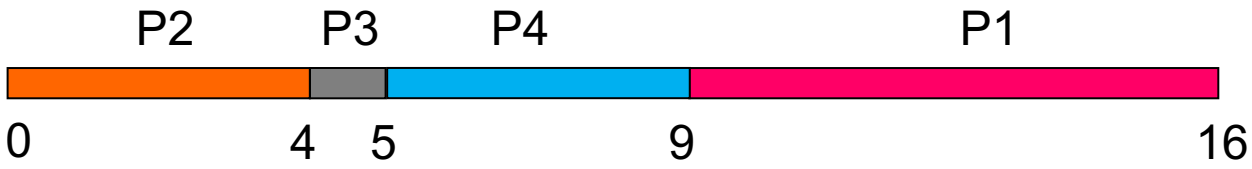
- Waiting time for P1=(11-2)=9, P2=1, P3=0, P4=(7-5)=2
- Average waiting time: (9+1+0+2)/4=3 (better than SJF, 3.75)

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority), it can be
  - Preemptive（upon new arrival of a higher priority process)
  - Non-preemptive
- Equal-priority processes are scheduled in FCFS order
- SJF is a special case of the general priority-scheduling algorithm, where priority is the inverse of predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Priority Scheduling: Example

Process	Burst Time	Arrival Time	Priority
P1	7	0	4
P2	4	0	1
P3	1	0	2
P4	4	0	3



- Waiting time for P1=9, P2=0, P3=4, P4=5
- Average waiting time: (9+0+4+5)/4=4.5

Q.1

- Response time vs. turnaround time

- **Answer**: The turnaround time is the sum of the periods that a process is spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O. Turnaround time essentially measures the amount of time it takes to execute a process.
- Response time, on the other hand, is a measure of the time that elapses between a request and the first response produced.
- The confusion could be, when we compute response time and turn around time in the examples of FCFS and SFJ, with a single CPU burst time of each process, both appear to be the same. However, we usually are only concerned with turn-around time in FCFS and SJF, and only consider response time in RR scheduling.

Q.2

- Waiting time vs. turnaround time

- **Answer**: For one singe CPU burst time, no matter what scheduling algorithm is used, turn-around time = CPU burst time (may be executed in multiple time, because of being pre-empted by high-priority process or quantum expiration) + waiting time.
- In another word, for a single CPU burst time, the turn-around time can be “easily” computed by the time the process completes its CPU burst time minus the arrival time (the time to join the ready queue initially).
- Waiting time = turn-around time – CPU burst time

Q.3

- Explain the process of starvation and how aging can be used to prevent it.

- **Answer**: Starvation occurs when a process is ready to run but is stuck waiting indefinitely for the CPU. This can be caused, for example, when higher-priority processes prevent low-priority processes from ever getting the CPU. Aging involves gradually increasing the priority of a process so that a process will eventually achieve a high enough priority to execute if it waited for a long enough period of time.


Q
A