**Task 1**

1. The ability to check input is a primary method of ensuring your program works properly without errors. Invalid input can cause even properly coded functions to fail, which can result in minor or major problems in program execution.
2. If functions are passing variables as parameters to other functions with input that is outside of the scope of the program, many programs can be caused to crash. That one bad input can have a domino effect and cause bad data to be stored and cause other bad input to be created for subsequent functions.

**Task 2**

1. As the busy waiting loop being removed, a thread shouted all 5 times, and then each of other threads took turn. On the contrary, when each thread yielded once after shouting, each of thread took turn to shout.
2. When input validation was disabled, no thread was created or ready for execution as the input started with characters, such as Ab143, @fdal5.53, A5-45, and etc. As input starting with numbers, such as 3.14, 6.1.1.1, 4Abc-1, and etc., the numbers before the first character were extracted and considered as integer input.

**Task 3**

Comparing a character to its associated Unicode value can also solve this problem. By checking each index of your char array if its greater than or equal to ‘a’ and less than or equal to ‘z’, this would identify any lower case letters within your char array. You would also have to check for their uppercase counterparts as well. The process for numbers is also the same, you would just check if your char array index is >= 0, and <= 9. For special cases such as ‘-‘ or ‘.’, you can just directly check to see if they are within the char array and set your flags accordingly. As you are processing the char array through the Unicode checks you would simply set flags for what is being identified, and once finished report the results of the input.

**Task 4**

1. For task 1, the input string is extracted from keyboard using *scanf()*. Upon passing the input string to *atoi()*, any characters which are not numbers (i.e. 0-9) are truncated. If so, each digit of the input string is checked. A flag will be raised, indicating the input is a character string, if anyone of the following four cases is true: a) if the first position is decimal point; b) there are more than 1 decimal points existing; c) ‘+’ or ‘-‘ appears in the middle of string (rather than the beginning of string); d) any digit is not ‘0-9’, ‘+’, ‘-‘, or ‘.’. When flag is not raised, the first position is checked to determine the sign of the decimal number. If nothing is truncated by *atoi()*, the input is a integer. The first position of the string is checked to determine the sign of integer. Resorting to this algorithm, all prompted consecutive inputs can be correctly categorized as required.

For task 2, the function *NumShouter()* and *Occurrence()* are defined to prompt the number of shouter and the amount of times each thread should shout respectively. The input type is checked by calling the function *CheckType()* defined in task 1. An *int* type is returned when input is valid. Otherwise, error message is sent and it prompts the user for input again. *Shout()* picks a random shout for each shouter and then yields CPU for random amount of cycles (i.e. between 2 and 5) after each shout. Continue looping the process until the required amount of shouts is completed. At the beginning of the program flow, the function *ThreadTest()* calls the function *NumShouter()* and *Occurrence()* prompting input. Upon returning the correct input, a *for* loop is utilized to create the corresponding number of threads. Then a tread is forked to the function *Shout()* once for each shouter.

For task 3, a global variable *selectArgs* is defined, whose value are determined by the prompted value associated with ‘-A’ option. As *selectArgs* is 1, task 1 is selected, while task 2 is selected as *selectArgs* is 2. Otherwise, an error message is sent indicating invalid input when ‘-A’ option is used. When ‘-A’ option is not used, ‘Opps!” is printed.

1. We learned how to use the random number generator in Nachos, how to fork multiple threads, how to simulate busy-wait loops, how to interpret command line functions, and how to thoroughly test input for validity.
2. The algorithm used in Task 1 is described as the pseudo-code as follow.

*str* = getInput; // use scanf()

*i\_str* = atoi(str); // only keep the first several number

if (strcmp(*str*, *i\_str*)==0) // nothing is truncated {

if (*str* == ‘0’) {*str* is zero;}

else if (*str*[0] == ‘-‘){*str* is a negative integer}

else {*str* is a positive integer}

} else { //something is truncated

scan each digit and raise *flag* if ***E*** is true; // refer to the description of ***E***

if (*flag* == 1){str is a character string}

else if (*str*[0] == ‘-’) {*str* is a negative decimal}

else {*str* is a positive decimal}

}

***E*** is true if

1) *str*[0] = ‘.’;

or 2) number of ‘.’in *str* is more than 1;

or 3) ‘+’ or ‘-‘ appears in *str* but not the beginning;

or 4) any digit is not ‘0-9’,’+’, ’-‘, or ’.’.

The algorithm used in Task 2 is described as the pseudo-code as follow.

*T* =NumShouter()// get checked user input for number of shouters and set it to T

*NumShout* = Occurrence()// get checked user input for number of shouts and set it to NumShout

for (*i*=0; *i*<*T*; *i*++) //loop to fork for each shout {

Thread \**t* = new Thread("forked thread");

t->Fork(Shout, i); //fork to shout()

}

*Shout()*{

Five random patterns are defined;

for (*i*=0; *i*<*NumShout*; *i*++) {//loop for number of shouts

j = Random() % 5;// chose random shout

printf(*pattern*[*j*]);

// yield CPU for random cycles before continuing

stallCycle = Random() % 3 + 2; // randomly choose between 2 and 5.

while (stallCycle != 0) {

currentThread->Yield();

stallCycle--;

}}}

The algorithm for task 3 is rather simple. Please refer to A1 in Task 4 for detail.