# CSCI 4730/6730
# (PA #1)

In Kee Kim

School of Computing

University of Georgia

# PA #1's Goal

- Fault-tolerant "word-count" program with multi-processor model
- **Part A:** Multi-process  a simplified version of "Word-Count" Program
- **Part B:** Fault Tolerance – Crash Handling

# What is Word Count?

- Count the total number of words in some documents.

- e.g., “the quick brown fox” “the fox ate the mouse” “how now brown cow”

- Total word counts = 13 (4 + 5 + 4)

# Base Code: `wc.c`

- Single Process for counting the number of words

# wc.c:main() function

```
 70: int main(int argc, char **argv)
 71: {
 72:   long fsize;
 73:   FILE *fp;
 74:   count_t count;
 75:
 76:   if(argc < 2) {
 77:     printf("usage: wc <filname>\n");
 78:     return 0;
 79:   }
 80:
 81:   count.linecount = 0;
 82:   count.wordcount = 0;
 83:   count.charcount = 0;
 84:
 85:   // Open file in read-only mode
 86:   fp = fopen(argv[1], "r");
 87:
 88:   if(fp == NULL) {
 89:     printf("File open error: %s\n", argv[1]);
 90:     printf("usage: wc <filname>\n");
 91:     return 0;
 92:   }
 93:
 94:   fseek(fp, 0L, SEEK_END);
 95:   fsize = ftell(fp);
 96:
 97:   count = word_count(fp, 0, fsize);
 98:   fclose(fp);
 99:
100:   printf("\n=======================================\n");
101:   printf("Total Lines : %d \n", count.linecount);
102:   printf("Total Words : %d \n", count.wordcount);
103:   printf("Total Characters : %d \n", count.charcount);
104:   printf("=======================================\n");
105:
106:   return(0);
107: }
```

```
  6: #include <stdio.h>
  7:
  8: typedef struct count_t {
  9:   int linecount;
 10:   int wordcount;
 11:   int charcount;
 12: } count_t;
```

```
-bash-4.1$ cat small.txt
abc
def
ghi
jkl
-bash-4.1$ ./wc small.txt
```

# wc.c:word_count() function

```
14: count_t word_count(FILE* fp, long offset, long size)
15: {
16:   char ch;
17:   long rbytes = 0;
18:
19:   count_t count;
20:
21:   // Initialize counter variables
22:   count.linecount = 0;
23:   count.wordcount = 0;
24:   count.charcount = 0;
25:
26:   printf("[pid %d] reading %ld bytes from offset %ld\n", getpid(), size, offset);
27:
28:   if(fseek(fp, offset, SEEK_SET) < 0) {
29:     printf("[pid %d] fseek error!\n", getpid());
30:   }
31:
32:   while ((ch=getc(fp)) != EOF && rbytes < size) {
33:     // Increment character count if NOT new line or space
34:     if (ch != ' ' && ch != '\n') { ++count.charcount; }
35:
36:     // Increment word count if new line or space character
37:     if (ch == ' ' || ch == '\n') { ++count.wordcount; }
38:
39:       // Increment line count if new line character
40:       if (ch == '\n') { ++count.linecount; }
41:       rbytes++;
42:   }
43:
44:   return count;
45: }
```

# Part A: Multi-process "Word-Count" Program

- The main problem of a single-process program (`wc.c`) is **scalability**. It cannot process a large number of words.
- To address the problem, you will convert the "word-count" program (`wc.c`) into the multi-process model (`wc_mul.c`).
- The main process creates multiple child processes and effectively divides work between child processes.
- The child process sends the result to the main process via Inter-process communication channel.
- We will use `pipe` in this assignment.
- The main process waits child processes, reads the result via IPC channel, and prints out the total on the screen.

# Part A: Multi-process "Word-Count" Program

- What you need to do
  - You will modify "`wc.c`" to create the multi-process model.
  - I recommend you to use the "`wc_mul.c`" template.
  - "`wc_mul.c`" needs three arguments
    - "`wc_mul.c`" receives the number of child processes and an input file name through the command-line argument.
    - 1st argument: # of child processes (default is 0)
    - 2nd argument: target file (input file)
    - 3rd argument: crash rate (default is 0%)
    - Example: `./wc_mul 4 large.txt 20`
  - **Please ignore 3rd parameter for Part A**

# Part A: Multi-process "Word-Count" Program

- Your submission for Part A
  - Explain your program structure and IPC in README.pdf file. **Only "pdf" format** will be accepted.
    - **Submitting in non-PDF format will result in a penalty.**
  - **To get full credit for part A, the multi-processing version of word-count should provide a significant performance improvement.**
  - **How much performance improvement can you get with 10 parallel processes?**
    - You will have a better idea once we finish chapter 4.
  - **We (technically TA ☺) will test the performance improvement!!!**

# Template: `wc_mul.c`

- Template for multi-process version of **`wc.c`**
- Optional – you can use your own version of **`wc_mul.c`**

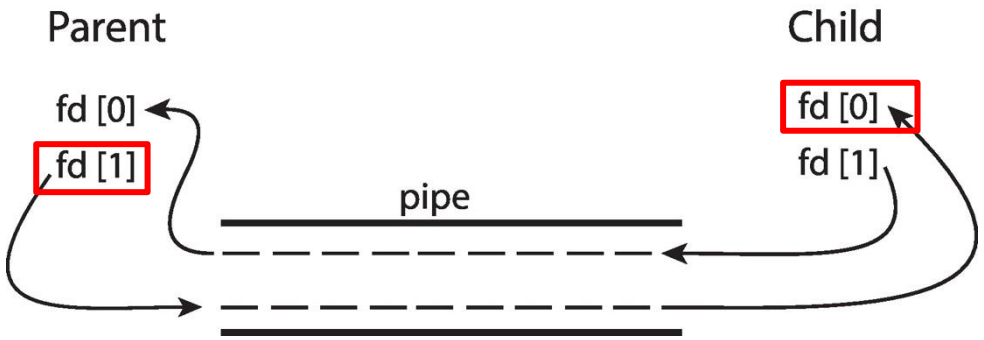
# wc_mul.c:main() function

```
 68: int main(int argc, char **argv)
 69: {
 70:   long fsize;
 71:   FILE *fp;
 72:   int numJobs;
 73:
 74:   //plist_t plist[MAX_PROC];
 75:   count_t total, count;
 76:   int i, pid, status;
 77:   int nFork = 0;
 78:
 79:   if(argc < 3) {
 80:     printf("usage: wc_mul <# of processes> <filname>\n");
 81:     return 0;
 82:   }
 83:
 84:   if(argc > 3) {
 85:     CRASH = atoi(argv[3]);
 86:     if(CRASH < 0) CRASH = 0;
 87:     if(CRASH > 50) CRASH = 50;
 88:   }
 89:   printf("CRASH RATE: %d\n", CRASH);
 90:
 91:   numJobs = atoi(argv[1]);
 92:   if(numJobs > MAX_PROC) numJobs = MAX_PROC;
 93:
 94:   total.linecount = 0;
 95:   total.wordcount = 0;
 96:   total.charcount = 0;
 97:
 98:   // Open file in read-only mode
 99:   fp = fopen(argv[2], "r");
100:
101:   if(fp == NULL) {
102:     printf("File open error: %s\n", argv[2]);
103:     printf("usage: wc <# of processes> <filname>\n");
104:     return 0;
105:   }
```

```
  6: #include <stdio.h>
  7: #include <stdlib.h>
  8: #include <sys/wait.h>
  9:
 10: #define MAX_PROC 100
 11: #define MAX_FORK 1000
 12:
 13: typedef struct count_t {
 14:   int linecount;
 15:   int wordcount;
 16:   int charcount;
 17: } count_t;
 18:
 19: /*
 20: typedef struct plist_t {
 21:   int pid;
 22:   int offset;
 23:   int pipefd[2];
 24: } plist_t;
 25: */
 26:
 27: int CRASH = 0;
```

# Recap: Ordinary Pipe

- Ordinary Pipes allow communication in producer-consumer style
- Producer writes to one end (**write-end** of the pipe)
- Consumer reads from the other end (**read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Parent-child relationship between communicating processes
- Windows calls these **anonymous pipes**



# wc_mul.c:main() function

```
 68: int main(int argc, char **argv)
 69: {
 70:   long fsize;
 71:   FILE *fp;
 72:   int numJobs;
 73:
 74:   //plist_t plist[MAX_PROC];
 75:   count_t total, count;
 76:   int i, pid, status;
 77:   int nFork = 0;
 78:
 79:   if(argc < 3) {
 80:     printf("usage: wc_mul <# of processes> <filname>\n");
 81:     return 0;
 82:   }
 83:
 84:   if(argc > 3) {
 85:     CRASH = atoi(argv[3]);
 86:     if(CRASH < 0) CRASH = 0;
 87:     if(CRASH > 50) CRASH = 50;
 88:   }
 89:   printf("CRASH RATE: %d\n", CRASH);
 90:
 91:   numJobs = atoi(argv[1]);
 92:   if(numJobs > MAX_PROC) numJobs = MAX_PROC;
 93:
 94:   total.linecount = 0;
 95:   total.wordcount = 0;
 96:   total.charcount = 0;
 97:
 98:   // Open file in read-only mode
 99:   fp = fopen(argv[2], "r");
100:
101:   if(fp == NULL) {
102:     printf("File open error: %s\n", argv[2]);
103:     printf("usage: wc <# of processes> <filname>\n");
104:     return 0;
105:   }
```

```
  6: #include <stdio.h>
  7: #include <stdlib.h>
  8: #include <sys/wait.h>
  9:
 10: #define MAX_PROC 100
 11: #define MAX_FORK 1000
 12:
 13: typedef struct count_t {
 14:   int linecount;
 15:   int wordcount;
 16:   int charcount;
 17: } count_t;
 18:
 19: /*
 20: typedef struct plist_t {
 21:   int pid;
 22:   int offset;
 23:   int pipefd[2];
 24: } plist_t;
 25: */
 26:
 27: int CRASH = 0;
```

# of child processes

# wc_mul.c:main() function (II)

```
107:   fseek(fp, 0L, SEEK_END);
108:   fsize = ftell(fp);
109:
110:   fclose(fp);
111:   // calculate file offset and size to read for each child
112:
113:   for(i = 0; i < numJobs; i++) {
114:     //set pipe;
115:
116:     if(nFork++ > MAX_FORK) return 0;
117:
118:     pid = fork();
119:     if(pid < 0) {
120:       printf("Fork failed.\n");
121:     } else if(pid == 0) {
122:       // Child
123:       fp = fopen(argv[2], "r");
124:       count = word_count(fp, 0, fsize);
125:       // send the result to the parent through pipe
126:       fclose(fp);
127:       return 0;
128:     }
129:   }
```

What is this for?

# wc_mul.c:main() function (II)

```
107:    fseek(fp, 0L, SEEK_END);
108:    fsize = ftell(fp);
109:
110:    fclose(fp);
111:    // calculate file offset and size to read for each child
112:
113:    for(i = 0; i < numJobs; i++) {
114:      //set pipe;
115:
116:      if(nFork++ > MAX_FORK) return 0;
117:
118:      pid = fork();
119:      if(pid < 0) {
120:        printf("Fork failed.\n");
121:      } else if(pid == 0) {
122:        // Child
123:        fp = fopen(argv[2], "r");
124:        count = word_count(fp, 0, fsize);
125:        // send the result to the parent through pipe
126:        fclose(fp);
127:        return 0;
128:      }
129:    }
```

- Now you know the total file size
- Now you know # children procs
- Divide workload equally to each child

- You may need

```
for (…)
    # offset for reading…
    plist[].offset = xyz;
```

# wc_mul.c:main() function (II)

```
107:    fseek(fp, 0L, SEEK_END);
108:    fsize = ftell(fp);
109:
110:    fclose(fp);
111:    // calculate file offset and size to read for each child
112:
113:    for(i = 0; i < numJobs; i++) {
114:      //set pipe;
115:
116:      if(nFork++ > MAX_FORK) return 0;
117:
118:      pid = fork();
119:      if(pid < 0) {
120:        printf("Fork failed.\n");
121:      } else if(pid == 0) {
122:        // Child
123:        fp = fopen(argv[2], "r");
124:        count = word_count(fp, 0, fsize);
125:        // send the result to the parent through pipe
126:        fclose(fp);
127:        return 0;
128:      }
129:    }
```

**pipe(plist[…].pipefd);**
→ you need error handling here.

# wc_mul.c:main() function (II)

```
107:    fseek(fp, 0L, SEEK_END);
108:    fsize = ftell(fp);
109:
110:    fclose(fp);
111:    // calculate file offset and size to read for each child
112:
113:    for(i = 0; i < numJobs; i++) {
114:      //set pipe;
115:
116:      if(nFork++ > MAX_FORK) return 0;
117:
118:      pid = fork();
119:      if(pid < 0) {
120:        printf("Fork failed.\n");
121:      } else if(pid == 0) {
122:        // Child
123:        fp = fopen(argv[2], "r");
124:        count = word_count(fp, 0, fsize);
125:        // send the result to the parent through pipe
126:        fclose(fp);
127:        return 0;
128:      }
129:    }
```

you need to change or not?

# wc_mul.c:main() function (II)

```
107:    fseek(fp, 0L, SEEK_END);
108:    fsize = ftell(fp);
109:
110:    fclose(fp);
111:    // calculate file offset and size to read for each child
112:
113:    for(i = 0; i < numJobs; i++) {
114:      //set pipe;
115:
116:      if(nFork++ > MAX_FORK) return 0;
117:
118:      pid = fork();
119:      if(pid < 0) {
120:        printf("Fork failed.\n");
121:      } else if(pid == 0) {
122:        // Child
123:        fp = fopen(argv[2], "r");
124:        count = word_count(fp, 0, fsize);
125:        // send the result to the parent through pipe
126:        fclose(fp);
127:        return 0;
128:      }
129:    }
```

```
ssize_t write(int fd, const void *buf, size_t count);
```

# wc_mul.c:main() function (II)

```
107:    fseek(fp, 0L, SEEK_END);
108:    fsize = ftell(fp);
109:
110:    fclose(fp);
111:    // calculate file offset and size to read for each child
112:
113:    for(i = 0; i < numJobs; i++) {
114:      //set pipe;
115:
116:      if(nFork++ > MAX_FORK) return 0;
117:
118:      pid = fork();
119:      if(pid < 0) {
120:        printf("Fork failed.\n");
121:      } else if(pid == 0) {
122:        // Child
123:        fp = fopen(argv[2], "r");
124:        count = word_count(fp, 0, fsize);
125:        // send the result to the parent through pipe
126:        fclose(fp);
127:        return 0;
128:      }
129:    }
```

End of child process

# wc_mul.c:main() function (III)

```
133:
134:    // Parent
135:    // wait for all children
136:    // check their exit status
137:    // read the result of normalliy terminated child
138:    // re-crete new child if there is one or more failed child
139:    // close pipe
140:
141:    printf("\n========== Final Results ================\n");
142:    printf("Total Lines : %d \n", count.linecount);
143:    printf("Total Words : %d \n", count.wordcount);
144:    printf("Total Characters : %d \n", count.charcount);
145:    printf("=========================================\n");
146:
147:    return(0);
148: }
```

Please refer to “wait_pid.c”

# wc_mul.c:main() function (III)

```
133:
134:    // Parent
135:    // wait for all children
136:    // check their exit status
137:    // read the result of normalliy terminated child
138:    // re-crete new child if there is one or more failed child
139:    // close pipe
140:                ssize_t read(int fd, void *buf, size_t count);
141:    printf("\n========== Final Results ================\n");
142:    printf("Total Lines : %d \n", count.linecount);
143:    printf("Total Words : %d \n", count.wordcount);
144:    printf("Total Characters : %d \n", count.charcount);
145:    printf("=========================================\n");
146:
147:    return(0);
148: }
```

After read the result from children procs,

- Sum all results
- Manage children procs (e.g., reset pid and/or offset)

# References for Part A

- Ordinary Pipe, Wait & Wait Pid
  - Last lecture

# Part B: Crash-handling

- If one or more child processes crash before they complete the job (before sending the result through pipe), the final output will be ***incorrect***.
- Therefore, we will need to monitor the exit status of each child. If there is one or more child processes crashed, we will need to create new child processes to complete the job.
- The parent process should "wait" for all child processes and monitor their exit status (hint: use **`waitpid()`**).
- If an exit status of a child process is ***"abnormal termination" by a signal***, the parent process creates a new child process to re-do the incomplete job.
- You can use 3rd command-line arguments (integer between 1 and 50) to trigger crash. 50 means each child process has 50% chance to be killed abnormally by signal. 0 means 0% chance to crash.
- Explain how your program handles crash in **README.pdf** file.

# wc_mul.c:main() function (III)

```
133:
134:    // Parent
135:    // wait for all children
136:    // check their exit status
137:    // read the result of normalliy terminated child
138:    // re-crete new child if there is one or more failed child
139:    // close pipe
140:
141:    printf("\n========== Final Results ================\n");
142:    printf("Total Lines : %d \n", count.linecount);
143:    printf("Total Words : %d \n", count.wordcount);
144:    printf("Total Characters : %d \n", count.charcount);
145:    printf("=========================================\n");
146:
147:    return(0);
148: }
```

# Chap #3 – extra slides && waitpid.c

- *int* **WIFEXITED** *(int status)*
  - returns a nonzero value if the child process terminated normally with exit or _exit.
- *int* **WEXITSTATUS** *(int status)*
  - If **WIFEXITED** is true of *status*, this returns the low-order 8 bits of the exit status value from the child process.
- *int* **WIFSIGNALED** *(int status)*
  - returns a nonzero value if the child process terminated because it received a signal that was not handled.
- *int* **WTERMSIG** *(int status)*
  - If **WIFSIGNALED** is true of *status*, this returns the signal number of the signal that terminated the child process.

# Submission

- Submit a tarball file using the following command
- `%tar czvf p1.tar.gz README.pdf Makefile wc_mul.c`
- README.pdf file with:
  - Your name
  - Explain your design of multi-process structure and IPC.
  - Explain how your program handles crash.
- Your code should be compiled and tested on **`odin`** server (odin.cs.uga.edu).
- Submit a tarball through ELC.
- The late submission policy is applied. (> 48hours, zero)

# Note

- Two dataset are given.
  - small.txt
  - large.txt

- How to test performance improvement?
  - **`time ./wc_mul 1 large.txt 0`**
  - **`time ./wc_mul 10 large.txt 0`**

# Note

- Deadline
  - 11:59 pm, Feb/13/2026 (Friday)
- Office hours
  - TA (Raka): Thurs 11a-1p in Boyd 819
  - Instructor: Thurs 4:15p - 5:15p in BOYD 802
  - TA and instructor will not debug your code.
- Grading

| | Undergrad | Grads (MS/PHD) |
|---|---|---|
| Part A | 70 | 30 |
| Part B | 30 | 70 |
| Total | 100 | 100 |

# Questions?