Unix
Bash
C
GNU
Systems

Software Systems

Lectures Week 10

Systems Programming 1

(Files, Processes, and Communication)

Prof. Joseph Vybihal

Computer Science

McGill University

Week 10 Lecture 1

Sequential, Block, and Random Access Files

Readings: https://www.thoughtco.com/random-access-file-handling-958450 and http://www.dummies.com/programming/c/basics-of-sequential-file-access-in-c-programming/

Basic File Organization

- Stream
  - Defined as a contiguous series of bytes, such that, traversal of the file occurs only in one direction, from the beginning of the file to the end of the file, one byte at a time.
- Block
  - The file is understood to be composed of units of equal sized data stored randomly. All the information is structured similarly, in units of N-bytes. The file can be traversed in any direction.
- Line
  - Data is organized into unequal byte sized units. Each unit needs a terminating symbol. File traversal occurs only in one direction, from beginning to end, looking for these markers.

Basic File Terminology

- Sequential access files
  - Stream and Line files are examples.
  - The fundamental property is that these files are accessed in only one direction, from the beginning to the end one byte at a time.
- Random access files
  - Block file is an example.
  - The fundamental property is that a data unit can be accessed randomly. These files operate analogously to arrays of struct.

Basic File Types

- Text (like .txt) or Binary (like a.out)
  - Text files and compiled programs are examples of sequential access files.
  - We have already seen how to read, write and append to Text files. Our lecture will focus on CSV and RAF. A compiler course will cover reading and writing a.out files.
- CSV (comma separated vector)
  - Many file types fall under the Line organization technique: .csv .ini .json, to name a few.
- RAF (random access file)
  - Examples would include: databases, caches, and quick access files.

The CSV File

Common uses for CSV are:

- As a configuration file
- As a simple database file

Other files that are Line based and used for similar purposes include:

- INI as strictly a configuration file
- XML and JSON as either configuration or simple databases
- JSON has also been used as a temporary information transfer format

The CSV File

Format:

- Record
  - Defined to be a Line of data terminating with a carriage return character.
- Field
  - Defined to be a sequence of characters terminating with the comma character or the carriage return character.

Example:

User,Password,FirstName,LastName

Jvybihal,abc123,Joseph,Vybihal

Mary.our,xyzAb!,Mary,Lou

Notice that the carriage return character and the comma character become reserved words that cannot be used as data. Escape characters can be used to overcome this limitation.

Example

```
#include<stdio.h>   #include<stdlib.h>   // in reality these needs to be different lines

char buffer[2000];  // some large number to handle long lines
char user[100], passw[100], firstName[100], lastName[100]; // fields with large sizes
int bufferIndex=0;

FILE *csv = fopen("file.csv","rt");
if (csv == NULL) exit(1);

fgets(buffer,1999,csv);
while(!feof(csv)) {
   bufferIndex = nextField(buffer, bufferIndex, user);
   bufferIndex = nextField(buffer, bufferIndex, passw);
   bufferIndex = nextField(buffer, bufferIndex, firstName);
   bufferIndex = nextField(buffer, bufferIndex, lastName);

   fgets(buffer,1999,csv);
}
```

Example (cont.)

```
int nextField(char theBuffer[], int index, char theField[]) {
        // theBuffer and theField use call-by-reference
        // index uses call-by-value
    int x, y=0;

    for(x=index; theBuffer[x]!='\0' && theBuffer[x]!=','; x++) {
        theField[y] = theBuffer[x];
        y++;
    }

    theField[y] = '\0'; // terminate it like a string

    return x+1; // as the new buffer index
}
```

RAF and Binary

To be able to move randomly within a file each unit of information needs to have a standard size.

RAF files can be text or binary, however binary files have the advantage of being faster to process because data type conversions from internal storage and disk storage can be skipped.

We will look only at binary RAF files.

The RAF Binary File

Takes advantage of:

- The C struct statement
- The stdio.h commands fread, fwrite & fseek to read/write entire struct data structures in one action
- Since struct structures are of the same byte-size we can compute the distance:
  - struct STUD array[10];
  - struct STUD *p = array + (2 * sizeof(struct STUD);
    - Which is the same as saying: p = array[2];

Example

```
#include<stdio.h>    #include<stdlib.h>    // in reality these need to be different lines

struct STUD {
   char name[100];
   int age;
   float GPA;
};

struct STUD x; // assume we put data in this
FILE *p = fopen("file.raf", "wb"); // write binary,   "rb" to read, "ab" to append
if (p == NULL) exit(1);

fwrite(x, sizeof(struct STUD), 1, p); // destructive, overwrites previous file, since "wb"
```

fwrite(struct *, sizeof, int repeat, FILE *);
fread (struct *, sizeof, int repeat, FILE *);

These two commands, by themselves, are sequential.

Example

```
// Writing to a RAF database

#include<stdio.h>    #include<stdlib.h>    // in reality these need to be different lines

struct STUD {
   char name[100];
   int age;
   float GPA;
} students[100];   // assume populated with values

FILE *p = fopen("database.raf", "wb");
if (p == NULL) exit(1);

For(x=0; x<100; x++)   fwrite(students[x], sizeof(struct STUD), 1, p);

fclose(p);
```

Example

```
// Reading from a RAF database

#include<stdio.h>    #include<stdlib.h>    // in reality these need to be different lines

struct STUD {
   char name[100];
   int age;
   float GPA;
} students[100];

int x = 0;
FILE *p = fopen("database.raf", "rb");
if (p == NULL) exit(1);

do {fread(students[x], sizeof(struct STUD), 1, p); x++;}   while(!feof(p));

fclose(p);
```

Random Access

The fseek command permits random motion within a file.

Syntax:

- int fseek(FILE *stream, long offset, int whence);
- void rewind(FILE *stream);

Where:

- Whence = SEEK_SET jump from beginning of file
- Whence = SEEK_CUR jump from current position in file
- Whence = SEEK_END jump from the end of the file
- Jumps are measured in OFFSET bytes. Positive numbers move forward through the file, negative numbers backwards.

Example

```
fseek(fp, 100, SEEK_SET);      // seek to the 100th byte of the file
fseek(fp, -30, SEEK_CUR);      // seek backward 30 bytes from the current pos
fseek(fp, -10, SEEK_END);      // seek to the 10th byte before the end of file

fseek(fp, 0, SEEK_SET);        // seek to the beginning of the file
rewind(fp);                    // seek to the beginning of the file
```

Question

If I have a database of students, struct STUD, and I would like to load into memory the 10^{th} student, how would I do this?

Question

Suggest how we might do this:

I have a sorted RAF of students, STRUCT STUD, sorted by student's last name. I want to find someone by their last name quickly. Assume last names are unique.

Week 10 Lecture 2

Inter-process Communication 1

Readings: http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

What is a process?

Program

- A program is understood to mean a file on disk containing a runnable algorithm (compiled or interpret-able)

Process

- A copy of the program but in RAM (the computer's live memory) and it is currently being executed

Program & Process

PROGRAM

- Loader
- Static Data
- Algorithms
- Libraries
- DLL requests

Tells OS what resources program needs to run

DLL libraries

Are attached to OS and shared with all processes

PROCESS

- Run-time Heap
- Run-time Stack
- Static Data
- Algorithms
- Libraries
- DLL requests

Local variables, recursion

malloc
calloc
new

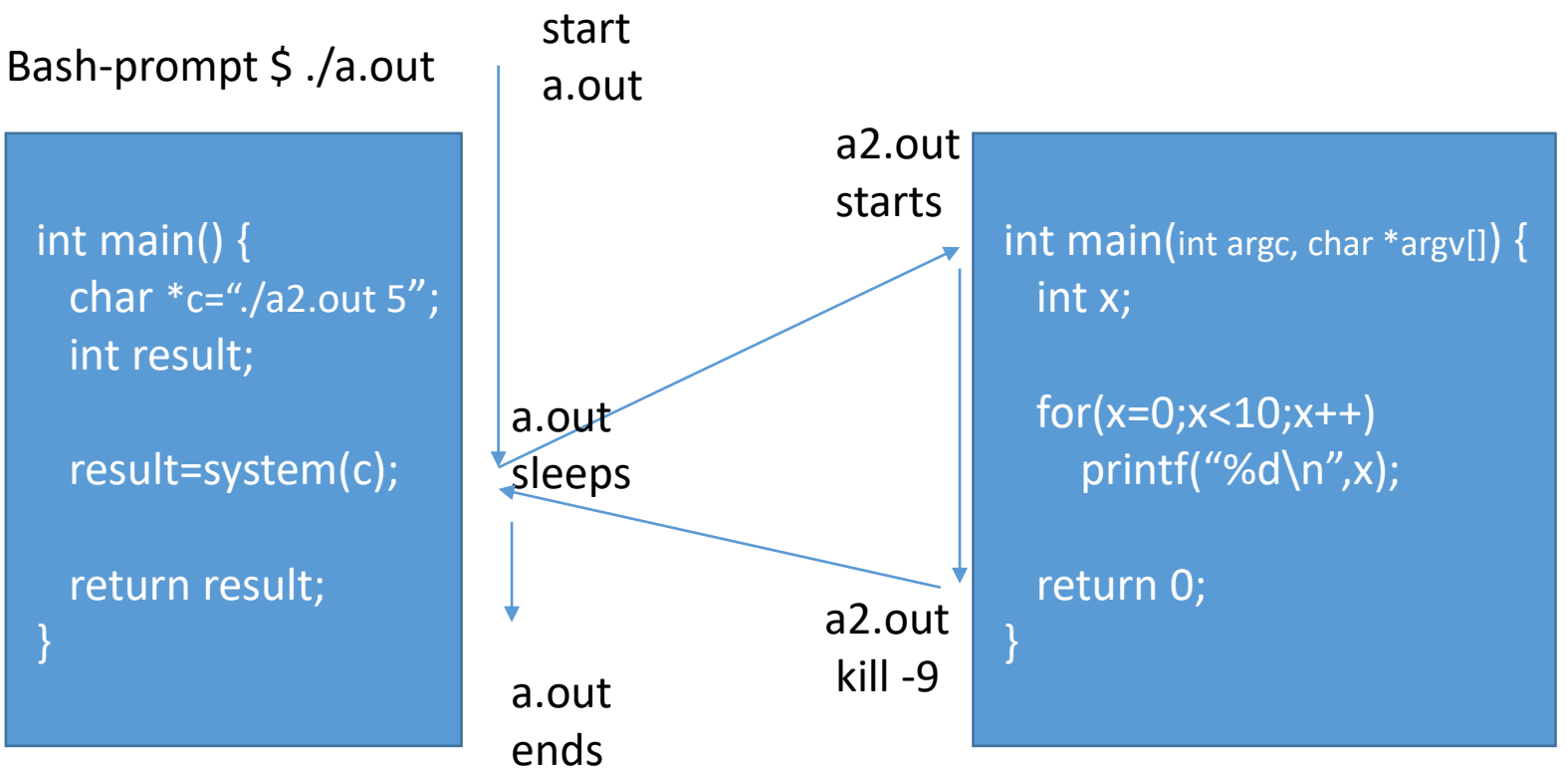
Three types of processes

- Shell processes
  - Is a process that runs within its own shell. Launching a process of this form will exit the current shell by launching a new shell and then using the command-line of the new shell to launch a program or execute a shell command.
- Cloned process
  - The currently executing process can clone itself. After the clone operation there are two identical processes running in the same shell concurrently.
- New process
  - The currently executing process can launch a new program within the current shell. Both processes run concurrently.

Three types of processes

- Shell processes
  - #include<stdlib.h>
  - int system(char *command_line_command);
  - Returns -1 on error, or return status number
- Cloned process
  - #include<unistd.h>
  - int fork();
  - Returns the process ID number
- New process
  - Not covered…

The system() function



Programs can use system() as often as needed.
A process invoked by system() can also use system(), recursive definition.

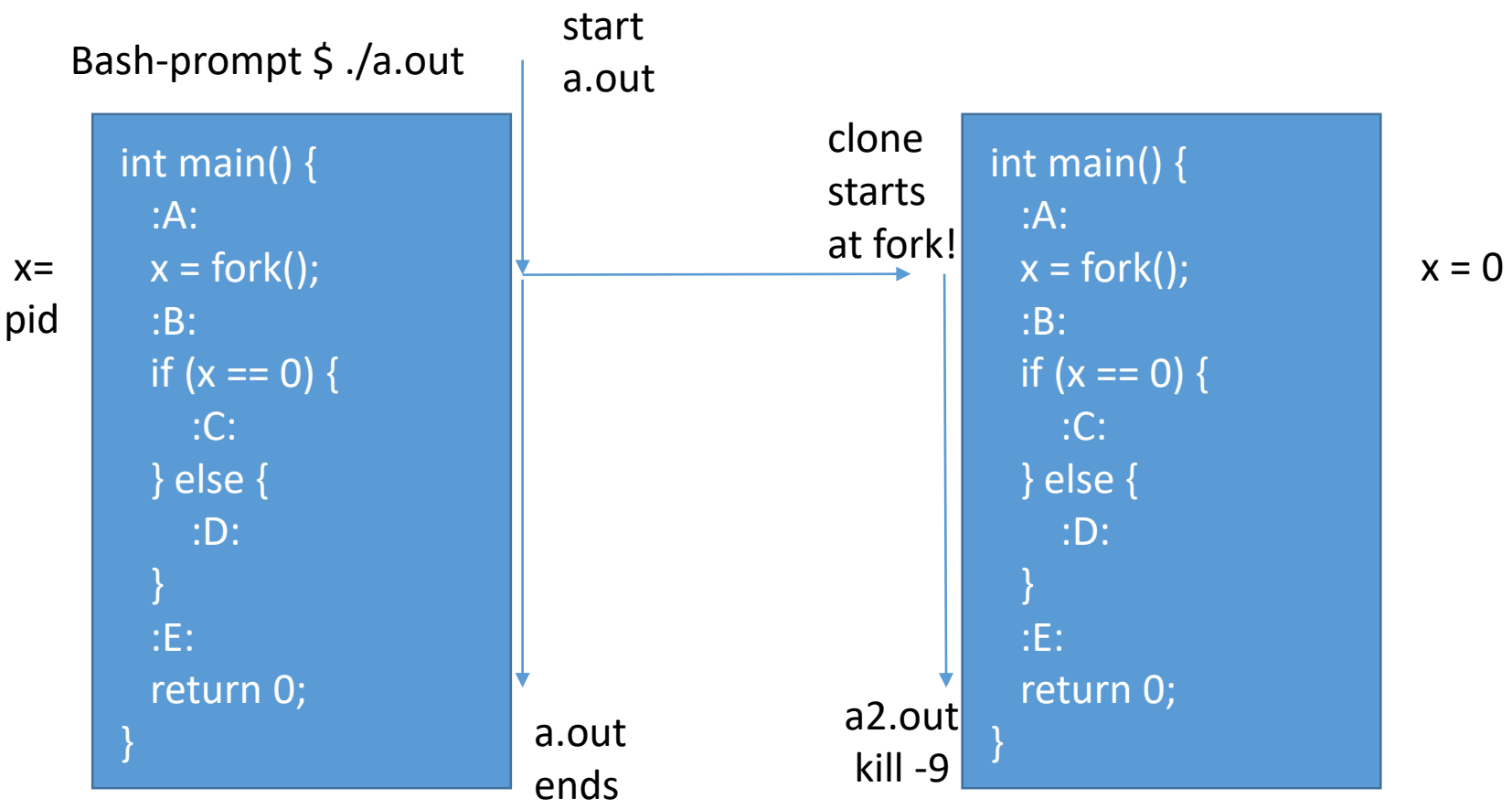
Notice how data is passed through command-line arguments & return statement.

The fork() function

```
#include <stdlib.h> /* needed to define exit() */
#include <unistd.h>/* needed for fork() and getpid() */
#include <stdio.h>  /* needed for printf() */

 int  main(int argc, char *argv[]) {
            int pid;    /* process ID */

            switch (pid = fork()) {
            case 0:                 /* fork returns 0 to the child */
                        printf("I am the child process: pid=%d\n", getpid());
                        break;

            default:    /* fork returns the child's pid to the parent */
                        printf("I am the parent process: pid=%d, child pid=%d\n", getpid(), pid);
                        break;

            case -1:    /* something went wrong */
                        perror("fork"); // send string to STDERR
                        exit(1);
            }
            exit(0);
}
```

Programs can use fork() as often as needed.
A process invoked by fork() can also use fork().

The fork() function



a.out → :A: :B: :D: :E:
clone → :B: :C: :E:
The clone receives a copy of all the variables (it is not shared).

The producer & consumer problem

The producer

- A program that generate data saving that data in a data structure or in a file.

The consumer

- A program that reads data from a data structure or a file performing some kind of computation.

Concurrency

- For some reason the producer and consumer need to run independently but cooperatively.
  - Maybe the computation is slow but the data comes in quickly. The intermediate data structure or file is used as a temporary cache.

Example

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

 int  main() {
            int pid = fork();
            if (pid == -1) exit(1);
            if (pid == 0) { producer();  wait(); }  // wait for consumer to end
            if (pid != 0) { consumer(); wait(); } // wait for producer to end
}

 void producer() {
            :
            :
 }

 void consumer() {
            :
            :
 }
```

Not an efficient example, but easy to understand.

Notice the use of functions to make the code easier to read.

Example

```
void producer() {
   char c = ' ';
   FILE *p;

   while (c != 'x') {                    // program stops at the input of 'x'
      c = getchar();

      while ((p=fopen("shared.txt","at")) == NULL) ; // notice semi-colon, we wait…
      fprintf(p,"%c\n", c);
      fclose(p);                         // give the other process a change to open the file
   }
}

void consumer() {
   char c; int pos = 0;
   FILE *q;

   do {
      c = removeCharacter(pos);   // we encapsulate further here for understandability
      printf("%c", c);
      pos++;                             // pos tracks the next data, pos is independent of producer
   } while (c != 'x');                   // program stops at input of 'x'
}
```

Data sharing is handled by a text file cache.

Example

```
char removeCharacter(int pos) {
   char c;
   FILE *q;
   int x = 0;

   while ((q=fopen("shared.txt", "rt") == NULL)  ;  // busy wait

   c=fgetc(q);
   while(!feof(q) && x < pos) {
            pos++;                                  // move to the correct position in file
            c=fgetc(q);
   }

   fclose(q);

   return c;                                        // return the character
}
```

How can we do this with fseek?

Notice that the two programs run independently from each other.
The speed by which they processed the file was different and did not matter.
However the current implementation assumes that producer is faster than consumer.

Question

The previous example assumed that producer was faster than consumer.

How could we modify consumer to handle the case of being sometimes faster than producer?

Using shared memory

```
#include <sys/types.h>
#include <sys/shm.h>

void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- Attach shared memory to program

```
int shmdt(const void *shmaddr);
```

- Detach shared memory from program

```
int shm_id = shmget(
        key_t    k,      /* the key for the segment      */
        int      size,   /* the size of the segment      */
        int      flag);  /* create/use flag              */
```

- Creates the shared memory space

Example

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/shm.h>
#include  <stdio.h>
    .....
 int      shm_id;      /* shared memory ID     */
 struct MEM { int a, b, c, status; } *p;   // status=0 empty, 1 filled
    .....
shm_id = shmget(IPC_PRIVATE, sizeof(struct MEM), IPC_CREAT | 0666);
if (shm_id < 0) exit(1);

/* now the shared memory ID is stored in shm_id */

 p = (int *) shmat(shm_id, NULL, 0);
 if (p == -1) exit(1);

 p->a = 5;
 p->status = 1;
 while(p->status == 1) sleep(1);  // wait for child, child will change status to 0
 exit(0);
```

IPC_CREAT | 0666 for a server (*i.e.*, creating and granting read and write access to the server) - Octal

0666 for any client (*i.e.*, granting read and write access to the client) - Octal

If a client wants to use a shared memory created with **IPC_PRIVATE**, it must be a child process

Child program is similar.
Examples: HERE and HERE.

Week 10 Lecture 3

Inter-process Communication 2

Shell-based concurrency

- The ampersand and semi-colon commands
  - Investigating is the ps command
- Shell memory
  - Before and after communication
  - Concurrent communication

Ampersand and semi-colon

- Sequential command-line execution
  - Example:
    - ls; cp file1.c /backup; cat file1.c
  - In the above example the ls command is executed first. After the ls command is finished the cp command executes. After cp is finished then the cat command runs last.
- Concurrent command-line execution
  - Example:
    - ls & cp file1.c /backup & cat file1.c
  - In this example all three commands execute concurrently.
  - They do not launch at the same time – if they run long enough they will all use the CPU at the same time.
  - Any concurrent output is displayed to the screen at the same time… mixed together.

Example

Bash-prompt $ vi producer.c

Bash-prompt $ vi consumer.c

Bash-prompt $ gcc –o producer producer.c

Bash-prompt $ gcc –o consumer consumer.c

Bash-prompt $ ./producer & ./consumer

These two programs runs concurrently without using fork().

These use a text file to coordinate and share the data (as we saw last class).

Example

```
// PRODUCER.C
int main() {
  char c = ‘ ‘;
  FILE *p;

  while (c != ‘x’) {
     c = getchar();

     while ((p=fopen(“shared.txt”,”at”) == NULL) ;
     fprintf(p,”%c\n”, c);
     fclose(p);
  }

  return 0;
}
```


```
// CONSUMER.C
int main() {
   char c; int pos = 0;
   FILE *q;

   do {
      c = removeCharacter(pos);
      printf("%c", c);
      pos++;
   } while (c != 'x');
}
char removeCharacter(int pos) {
   char c;
   FILE *q;
   int x = 0;

   while ((q=fopen("shared.txt", "rt") == NULL)  ;

   c=fgetc(q);
   while(!feof(q) && x < pos) {
            pos++;
            c=fgetc(q);
   }
   fclose(q);
   return c;
}
```

Accessing Shell Memory

Remember:

Bash-prompt $ set

The 'set' command displays all the variables defined within the shell memory.

C can access these variables:

```
#include <stdlib.h>
char *data = getenv("VARIABLE_NAME");
```

```
printf("PATH : %s\n", getenv("PATH"));
printf("HOME : %s\n", getenv("HOME"));
printf("ROOT : %s\n", getenv("ROOT"));

int n = atoi(getenv("CONTENT_LENGTH"));
```

The getenv() returns NULL if it failed to find the variable.

```
int setenv (const char *name, const char *value, int replace)
```

```
int success = setenv("VAR_NAME", "VALUE", 1);
```

Replace = 1 for true, 0 for false
Success = 0 for true, -1 for false

Example

Bash-prompt $ ./prog1; ./prog2

Notice that prog1 runs to completion first before prog2 starts.

In prog1:

- setenv("x", "yes", 1);

In prog2:

- char *p = getenv("x");

Notice that a preceding program can communicate with a following program.

Example

Bash-prompt $ set x="yes"

Bash-prompt $ ./prog2

User sets the variable at the command prompt

In prog2:

- char *p = getenv("x");

Notice that the user can change the "environment" causing prog2 to behave differently.

Example

Bash-prompt $ ./prog3 & ./prog4

Two concurrent programs can coordinate using the shell memory

Assume a shell variable TURN is “3” or “4”.

In prog3:

```
char *p;
while (1) {
   do { p = getenv(“TURN”); }
   while (strcmp(p, “3”) != 0) ; // wait
   … do something since it is 3 now …
   setenv(“TURN”, “4”, 1);
}
```

In prog4:

```
char *q;
while (1) {
   do { q = getenv(“TURN”); }
   while (strcmp(q, “4”) != 0) ; // wait
   … do something since it is 4 now …
   setenv(“TURN”, “3”, 1);
}
```

Question

Suggest a way we could use shell memory to do the producer/consumer problem. Use the ampersand to launch the two programs.

In this case we would like to simply produce one value and consume that one value, and repeat. The user enters values from the keyboard until they write the word DONE, which then terminates both programs.