CSCI 4061
Recitation 2
Today

- fork – exec – wait
- gdb
- Makefile
A Process

• What is a process in Unix?
  – A process is a program in execution
  – Difference between a process and program

• Different states
  – READY
  – RUNNING
  – WAITING
Process pool

• How do you find out what processes your system is running currently?
  • `ps -a`
  • `man ps`
fork()

- fork() – Creates a new process
- Parent process executes fork and creates an almost identical copy of itself
- Child process inherits parent’s state and context:
  - Code, data, open files
  - Program counter and stack
- #include <unistd.h>
- If fork() fails, it returns -1 and sets a errno to EAGAIN
- If fork() succeeds, it returns 0 to the child and the child’s pid to the parent.
- Potential pitfalls:
  - duplicate memory can provide intermixed output (try stdout)
parent_process(pid=1000)

call fork()

child_process(pid=2000)

call fork()

child PID=2000

parent_process(pid=1000)

printf("I am a parent");

child_process(pid=2000)

printf("I am a child");

child PID=0

Error!
fork_ex.c

```c
pid_t childpid;

childpid = fork();
if (childpid == -1)
{
    perror("fork() failed");
    return 1;
}
if (childpid == 0)
    printf("I am a child with id %ld\n", (long)getpid());
else
    printf("I am a parent with id %ld\n", (long)getpid());
return 0;
```
wait()

- When a process creates a child, both parent and child proceed execution from the point of `fork()`
- The parent can execute `wait()` or `waitpid()` to block until the child executes
  - `wait()` : waits for the termination of one of the children
  - `waitpid()` : waits for the termination for specified child process
wait_ex.c

```c
pid_t childpid;
pid_t waitreturn;
int status;
childpid = fork();

if(childpid==1)
{  
    perror("fork");
    exit(0);
}
else if(childpid==0){
    printf("I am a child\n");
    exit(3);
}
else {
    waitreturn = wait(&status);
    if(WIFEXITED(status)) {
        printf("child exited with status %d\n",WEXITSTATUS(status));
    }
}
```
exec()

- exec – execute a shell command or program
- `Int execv(const char *path, char *const argv[])`;
- Six of them – `execl, execlp` and `execle` form one family while `execv, execvp` and `execve` form the other
- `man` them all – On your own time!
execl_ex.c

```c
pid_t childpid;
    childpid = fork();
    if(childpid== -1){
        perror("Failed to fork");
        return 1;
    }
    // child code
    if(childpid == 0){
        execl("/bin/ps", "ps", "-af", NULL);
        perror("child failed to exec all_ids");
        return 1;
    }
    if(childpid != wait(NULL)){
        perror("parent failed to wait due to signal or error");
        return 1;
    }
```
Debugging using GDB

- GDB (GNU Debugger) is the standard command-line debugger for the GNU operating system

<table>
<thead>
<tr>
<th>How do I?</th>
<th>GDB commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile with debugging symbols</td>
<td>gcc -c -o helloWorld helloWorld.c</td>
</tr>
<tr>
<td>Run programs in GDB</td>
<td>gdb ./helloWorld</td>
</tr>
<tr>
<td>Restart program in debugger</td>
<td>kill run</td>
</tr>
<tr>
<td>Exit debugger</td>
<td>quit</td>
</tr>
</tbody>
</table>

- Compile source files with -g option to enable debugging
  - -g option tells compiler to put debug information in the object file
  - cc -g -o HelloWorld HelloWorld.c
  - gdb ./HelloWorld
Executing GDB

- GDB allows –
  - To execute and stop your program at specified points
  - Examine what has happened and inspect your program after stop-point
  - Make changes to variables and run

<table>
<thead>
<tr>
<th>How do I?</th>
<th>Gdb Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Restart execution</td>
<td>continue</td>
</tr>
<tr>
<td>See where the program stopped</td>
<td>list</td>
</tr>
<tr>
<td>Step through code line-by-line</td>
<td>Next step</td>
</tr>
<tr>
<td>Examine variables</td>
<td>print</td>
</tr>
</tbody>
</table>
## GDB Stack-breakpoints-watchpoints

<table>
<thead>
<tr>
<th>How do I?</th>
<th>Gdb Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get a backtrace</td>
<td>backtrace</td>
</tr>
<tr>
<td>Change stack frame</td>
<td>frame</td>
</tr>
<tr>
<td>Set breakpoint on line/function</td>
<td>Break line_number/function_name</td>
</tr>
<tr>
<td>Set watchpoint on variable</td>
<td>Watch variable</td>
</tr>
<tr>
<td>Get list of breakpoints/watchpoints</td>
<td>Info breakpoints</td>
</tr>
<tr>
<td>Disable breakpoints</td>
<td>Disable/clear breakpoints</td>
</tr>
</tbody>
</table>

To debug programs with multiple processes, use `set follow-fork-mode mode` where `mode` can be child or parent.
Debugging fork_ex.c

1. Compile fork_ex.c with –g option: gcc –g –o fork_ex fork_ex.c
2. Start GDB with a.out: gdb a.out
3. Set breakpoint in main function: break main
4. Check all breakpoints: info breakpoints
5. Run the program: run
6. Do “next” and observe the prints
7. To switch between parent and child processes:
   set follow-fork-mode mode where mode can be child or parent
Makefile

• What is the utility of makefiles
  – You don’t have to compile each file in your project separately
  – When you change only a subset of the files in your project, you don’t have to compile all the files.
  – Makefiles take care of compiling only the changed files

• Makefile contains instructions and rules to compile the source-files in your project and build executables
Makefile

• ‘make’ is a utility which reads the makefiles and builds target/executables according to the rules specified in makefile

• Type – `make -f <name of makefile>`
  – If you don’t use option -f, by default make reads the file with name ‘makefile’
Makefile

• Rules
  – A rule tells when and how to make a target e.g. to compile a source file or build an executable
  – Rules syntax –
    
    target : prerequisites
    command

    helloworld : helloworld.o
       gcc -o helloworld helloworld.o
    helloworld.o : helloworld.c
       gcc -w -c helloworld.c

  – Here helloworld.c will be compiled only when modification time of helloworld.c is more recent than helloworld.o
#the C compiler
CC = gcc
CFLAGS = -g

#this rule is to link the object code into an executable
helloworld: helloworld.o
   $(CC) -o helloWorld helloWorld.o

#this rule is to compile the source code
helloworld.o: helloworld.c
   $(CC) $(CFLAGS) -c helloWorld.c

clean:
   rm -f *.o
   rm -f helloWorld
Questions?