Creating Concurrent Processes

Lecture 2

February 11, 2025

To Dos

Reading for next time

Program 1

Office hours (M 2-3:30, W 1-2:30)

Program 1

Part 1

- Look for strings in dictionary that start with any combination of specified 3 letters
- Divide work among 6 concurrent, child processes
- Child processes send results to parent
- Parent prints results after all children finish

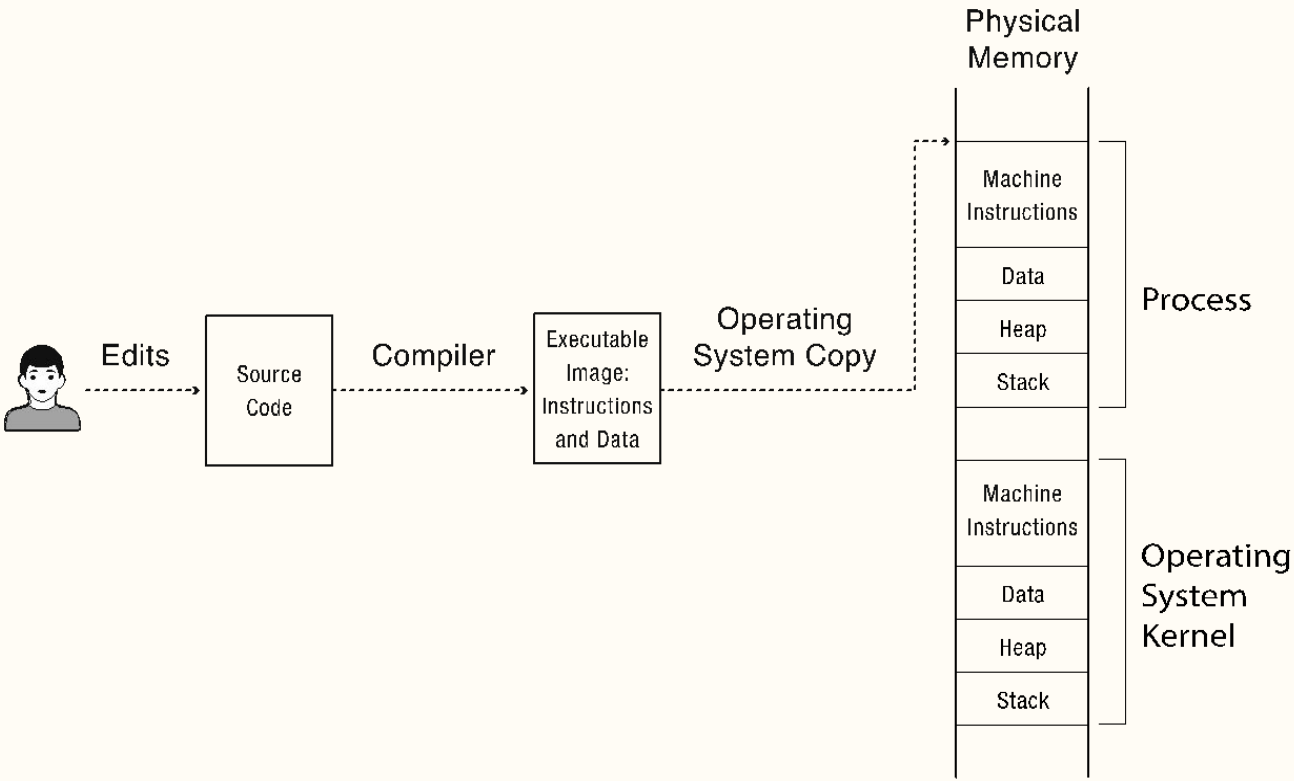
Part 2

- Explore graph algorithm written in C++ from GAP benchmark suite
- Understand high level goal of algorithm
- Use gprof to understand where time is spent and call graph
- Use debugger and code examination to understand time-intensive portions
- Algorithms:
    - Breadth-First Search (BFS)
    - Single-Source Shortest Paths (SSSP)
    - PageRank (PR_SPMV)
    - Connected Components (CC_SV)
    - Betweenness Centrality (BC)
    - Triangle Counting (TC)

Main Points

- Creating and managing processes
  - fork, exec, wait
- Performing I/O
  - open, read, write, close
- Communicating between processes
  - pipe, dup, select, connect

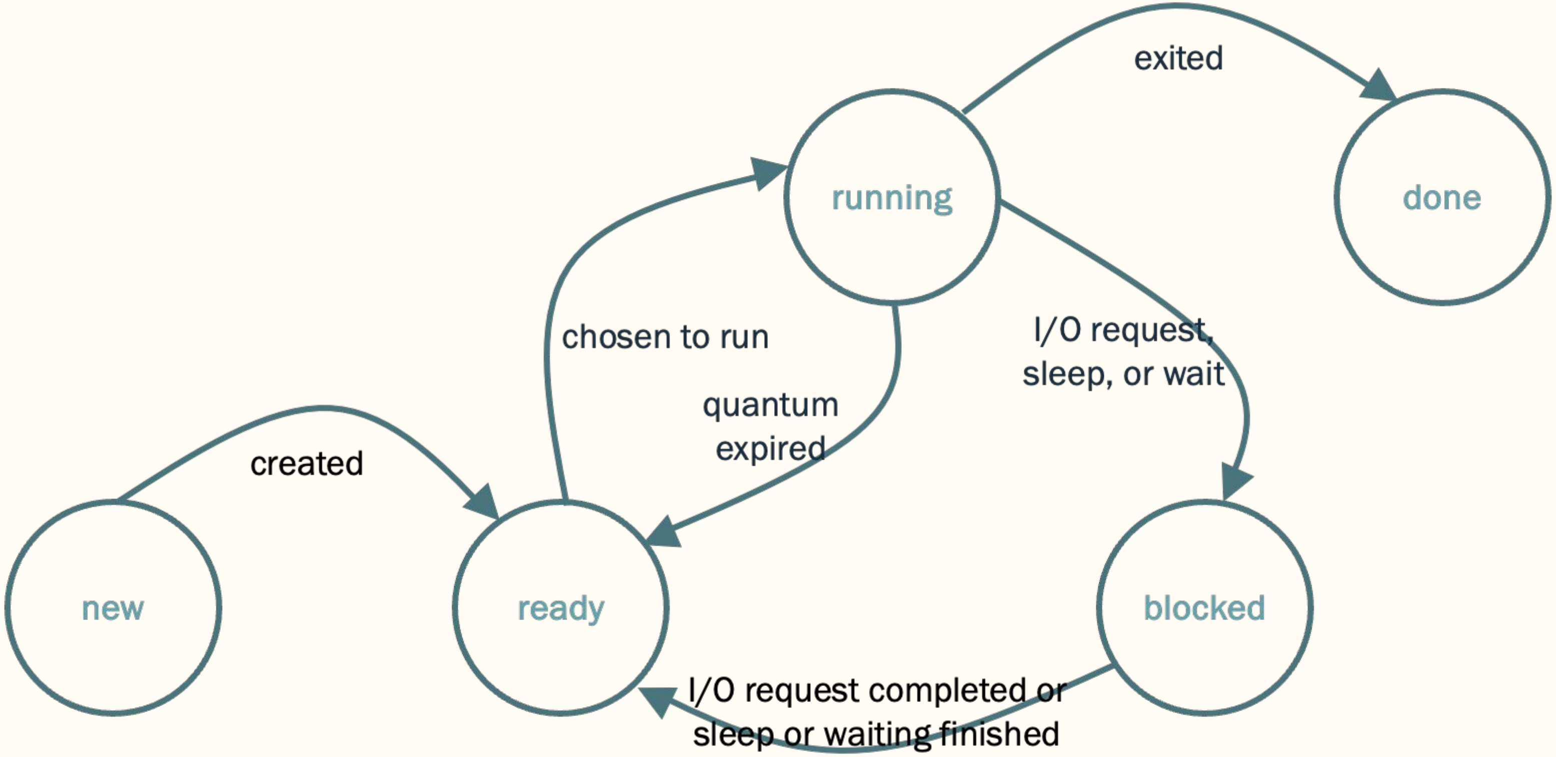
Program to Process



Process State (i.e., Process Context)

- Address space
- Instruction Pointer (IP/PC)
- Register state
- Status of program I/O (e.g. files)
- User and process ID
  - Process ID (PID)
  - Parent Process ID (PPID)
- Current state (i.e. new, running…)
- Accounting info
- Privileges
- Memory management info

Process States



Process States

- New – being created (not yet runnable)
- Ready – Eligible for running. Just waiting for OS to give it the processor
- Running – OS has chosen this process to run and it is running on processor
- Blocked – Not eligible for running because it's waiting for I/O, another process to signal it (e.g. wait()), or waiting for sleep() call to finish
- Done – finished executing correctly or abnormally

Unix Processes

- Process ID
  - getpid()

- Parent process ID
  - getppid()

- Permissions for process come from user account and its group. Effective IDs typically same as read IDs (but can be changed)
  - getuid() and geteuid()
  - getgid() and getegid()

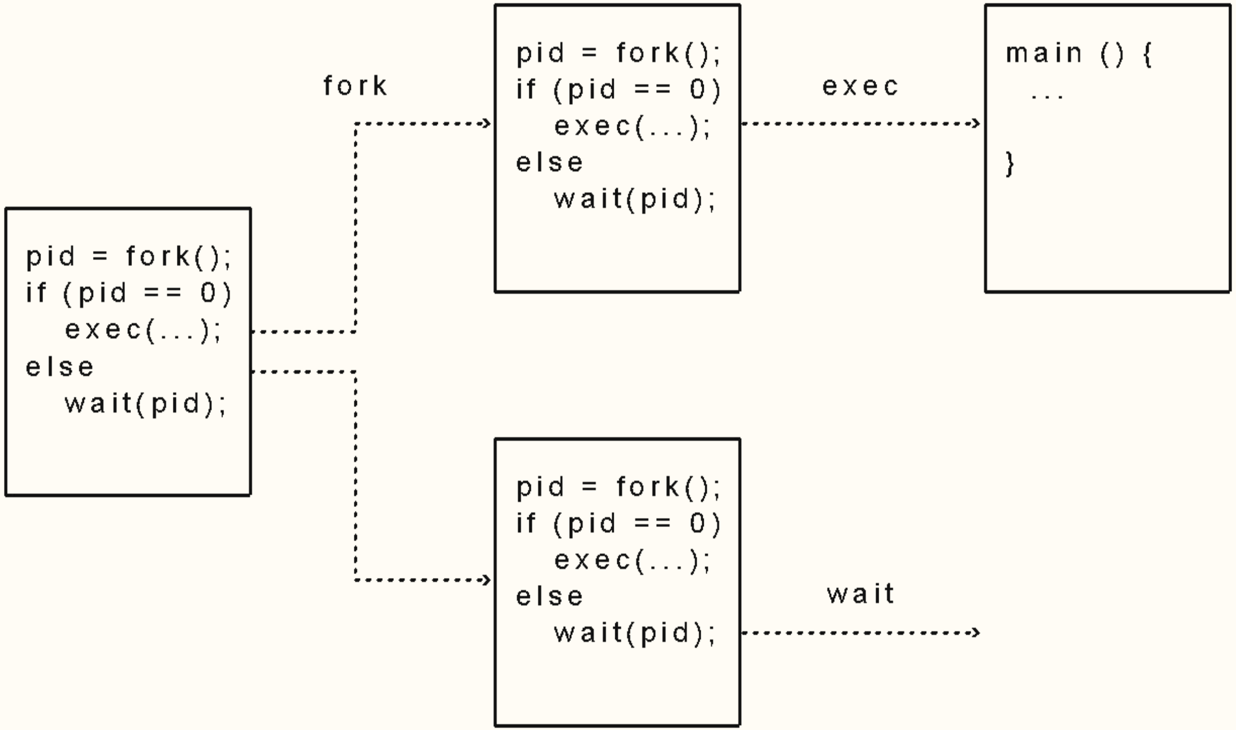
Simple Process Example

- Use `ps -al` to see all running process

UNIX Process Management

- UNIX fork – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX exec – system call to change the program being run by the current process
- UNIX wait – system call to wait for a process to finish
- UNIX signal – system call to send a notification to another process

UNIX Process Management



Unix `fork()`

- Creates new process
- New process has exact copy of memory image, same next instruction address
- Inherits open files, environment, and privileges from parent
- Call returns
  - 0: to child process
  - Child pid: to parent process
  - -1: error

Question: What does this code print?

```
int child_pid = fork();
if(child_pid == -1){
    printf("Error creating process \n");
    exit(1);
}
else if (child_pid == 0) {
    printf("I am process #%d\n", getpid());
    return 0;
} else {
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Question: What does this code print?

```
int child_pid = fork();
if(child_pid == -1){
    printf("Error creating process \n");
    exit(1);
}
else if (child_pid == 0) {
    sleep(5);
    printf("I am process #%d\n", getpid());
    return 0;
} else {
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

`wait()`, Zombies, and Orphans

- A process and its children will finish at different times
  - They will compete for same processor resources
- Parent process can wait for children to complete with `wait()` and `waitpid()`
- Zombie process
  - Child process completes and parent hasn't called wait on it yet
- Orphan
  - Parent process completes and didn't call wait() on child
  - Child process' parent because system `init` process (pid==1)

Simple wait for child to finish example

```
int main(int argc, char **argv)
{
  int child_pid = fork();
  if(child_pid == -1){
    printf("Error creating process \n");
    exit(1);
  }
  else if (child_pid == 0) {
    sleep(5);
    printf("I am process %d\n", getpid());
    return 0;
  } else {
    wait(NULL);
    printf("I am parent %d of process %d\n", getpid(), child_pid);
    return 0;
  }
}
```

Running a different program: `exec()`

- Process can begin running different program via exec() family of system calls
  - execl(), execle(), execlep() : when you know program path at compile time
  - execv(), execve(), execvp() : when you know program path at runtime
- Overwrites process image with specified program's image
  - Entire address space overwritten with new program
  - Register state overwritten with values for starting new program
- Should never return

Simple `execl()` example

```
int main(int argc, char **argv)
{
  int child_pid = fork();
  if(child_pid == -1){
    printf("Error creating process \n");
    exit(1);
  }
  else if (child_pid == 0) {
    printf("I am process %d calling date\n", getpid());
    execl("/bin/date", "date", NULL);
    printf("Code after execl -- shouldn't be reached\n");
    return 0;
  } else {
    wait(NULL);
    printf("I am parent %d of process %d\n", getpid(), child_pid);
    return 0;
  }
}
```

What if we had the following `fact` program?

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

int main(int argc, char **argv)
{
  if(argc > 1){
    int val = atoi(argv[1]);

    int result = 1;
    for(int i = 1; i <= val ; i++){
      result *= i;
    }
    printf("The factorial of %d is %d!\n", val, result);
  }
}
```

What will this code print?

```c
int main(int argc, char **argv)
{
  int val = 4;
  int pid = fork();
  if(pid == 0){
    val += 4;
    int child = fork();
    if(child == 0){
      char *str = "3";
      execl("fact", "fact", str, NULL);
      printf("val is %d\n", val);
    }
    else{
      val += 5;
      waitpid(child, NULL, 0);
      printf("val2 is %d\n", val);
    }
  }
  else{
    val += 6;
    waitpid(pid, NULL, 0);
    printf("val3 is %d\n", val);
  }
}
```

Processes are Independent

- Processes know about their parent and children
- But after creation, a process doesn’t share memory with any other process
- How can they collaborate and interact?
  - Must explicitly set up mechanism for sharing information
    - Files
    - Pipes and FIFOs
    - Signals
    - Other mechanisms:
      - Shared memory
      - Semaphores
      - Messages

UNIX I/O

- Uniformity
  - All operations on all files, devices, interprocessor communication use the same set of system calls: `open()`, `close()`, `read()`, `write()`
- Open before use
  - `open()` returns a handle (file descriptor) for use in later calls on the file
- Explicit close
  - To garbage collect the open file descriptor

Unix files

- Special files : represent devices and are located in /dev directory
  - Block special file : device w/ characteristics like a disk
    - Block sized transfers
  - Character special file : device with characteristics similar to a terminal
- Regular file : ordinary data file on disk

UNIX File System Interface

- UNIX file `open()` is a Swiss Army knife:
    - Open the file, return file descriptor
    - Options:
        - if file doesn’t exist, return an error
        - If file doesn’t exist, create file and open it
        - If file does exist, return an error
        - If file does exist, open file
        - If file exists but isn’t empty, nix it then open
        - If file exists but isn’t empty, return an error
        - …

Simple `open()` and `close()`

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  int filedes = open("foo.txt", O_RDWR);
  if(filedes == -1){
    printf("Error opening file\n");
  }
  else{
    printf("Opened file correctly %d\n", filedes);
    close(filedes);
  }
}
```

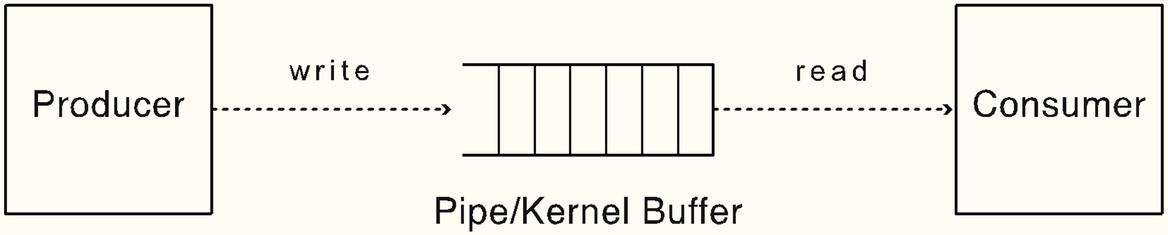
read() and write()

- `ssize_t read(int filedes, void*buf, size_t nbyte)`
  - Tries to read `nbytes` from `filedes` into `buf`
  - Returns number of bytes actually put into buffer
- `ssize_t write(int filedes, void*buf, size_t nbyte)`
  - Tries to write `nbytes` from `buf` into `filedes`
  - Returns number of bytes actually written into file

Files as a Mechanism for Sharing

- Processes can create, read, and write files assuming they have permission to do so
- One process can write to a file that another can subsequently read
  - What problems might arise?

UNIX pipe



- A pipe is a kernel buffer with two file descriptors, one for writing and one for reading
- Data is read in same order it is written
- Producer and consumer can work at different rates

UNIX Pipe

```
#include <unistd.h>

int pipe(int filedes[2]);
```

- Enables interprocess communication between **related** processes
  - Has to be inherited from parent process
- Represented as a special file
- Communication buffer accessed through 2 file descriptors
  - `filedes[0]` : reading
  - `filedes[1]` : writing
- Returns 0 on success, -1 on failure
- Can `read()`, `write()`, and `close()` on file descriptors
  - Reads/writes are blocking
  - If file descriptor closed, read() returns 0

Blocking

- Process waits until something becomes available for it to read on pipe or something is listening on pipe
- Can be switched to nonblocking using flags

Simple Pipe Example : wait_child.c

```
int main(int argc, char **argv)
{
  int fd[2];
  if(pipe(fd) == -1)exit(1);
  int child_pid = fork();
  if(child_pid == -1)exit(1);
  else if (child_pid == 0) {
    char buffer[100];
    if(read(fd[0], buffer, 100) != 0)
      printf("Received %s\n", buffer);
    printf("I am process %d\n", getpid());
    return 0;
  } else {
    char buffer[100];
    strcat(buffer, "Message");
    if(write(fd[1], buffer, 10) != 0)
      printf("Sent %s\n", buffer);
    wait(NULL);
    printf("I am parent %d of process %d\n", getpid(), child_pid);
    return 0;
  }
```

Example between parent and child: two_way_pc.c

Example solution to process hanging

Example one-way between two children: between_children.c

FIFOs: Communicating Between Unrelated Processes

```
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode)
```

- Named pipes represented by special files that persist after all processes close them
  - path : directory path and name of special file to create
  - mode : permissions
- To create, either
  - `mkfifo` at command prompt
  - Use function call in program
- Return 0 on success, -1 on failure
- Remove same way you remove a file (i.e. `rm` or `unlink()`)

File Permissions

Symbol	Meaning
S_IRUSR	Read by owner
S_IWUSR	Write by owner
S_IXUSR	Execute by owner
S_IRWXU	Read, write, execute by owner
S_IRGRP	Read by group
…	
S_IROTH	Read by other
…	
S_ISUID	Set user ID on execution
S_ISGID	Set group ID on execution

Example: create_fifo.c

Use FIFOs just like a file

- `open()`
  - `When pipe/fifo opened for reading, blocks until opened for writing`
- `read()/write()`
  - Writes to fifos of no more than PIPE_BUF bytes are atomic
    - Not true of reads
- `close()`

Producer/consumer example: reader.c and writer.c

- Demonstrate with 1-on-1
- Demonstrate with multiple instances
- Demonstrate with different modes

Example: client.c and server.c

Signals

- Software notification to a process of an event
- Generated when event that causes signal occurs
- Delivered when process takes action based on signal
  - Process installs signal handler to handle signal (via `sigaction()`)
    - Note: sigaction() can be set up to ignore or take default action instead of having handler
    - Note: Can block certain types of signals (via `sigprocmask()`)
- User can only send to process owned by it

Required Signals

Signal	Description
SIGABRT	Process abort
SIGALRM	Alarm clock
SIGBUS	Access undefined part of memory object
SIGCHLD	Child terminated, stopped, or continued
SIGCONT	Execution continued if stopped
SIGFPE	Error in arithmetic operation (e.g. div by 0)
SIGHUP	Hang-up on controlling terminal (process)
SIGILL	Invalid hardware instruction
SIGKILL	Terminated
SIGINT	Interactive attention signal (often Ctrl-C)
…	