Dave Sizer
8/3/15
CS370 Written Assignment 1

1. A program is an abstract collection of memory manipulation, arithmetic, and control flow operations used to accomplish some task with a computer. It can be implemented in any of a variety of programming languages. An executable is a program that has been assembled into machine language, an architecture-specific set of instructions that can be directly loaded into memory and executed on the CPU. A process is an abstraction maintained by the operating system that maintains all needed information about a single "task" being performed by the computer. One element of a process is the actual executable code, but the process contains a lot more information relevant to its handling and use in the context of an operating system, such as its memory footprint, all of its registers (these are used by the operating system for context switching), as well as the file handles it has open, the ports it is bound to, its current timeslice, etc.

2.

```
int main()
{
  // Create a shared memory space
  key_t key;
  pid_t *pids;
  int i;
  key = ftok("./370_wa1.c", 'a');
  if (key == -1) perror("ftok");

  i = shmget(key, 12, 0666 | IPC_CREAT); // 12 bytes = 3 pids
  if (i<0) perror("shmget");
  pids = shmat(i, NULL, 0);
  if (pids == -1) perror("shmat");
  bzero(pids,12);

  pids[0] = getpid();
  pid_t temp;

  if ((temp = fork()) == 0) // Son
  {
    if ((temp = fork()) == 0) // Grandson
    {
      printf("My pid is %d, my parent's pid is %d, and my grandparent's pid is %d!\n",
          pids[2], pids[1], pids[0]);
    }
    else
      pids[2] = temp;
  }
  else
    pids[1] = temp;


  shmdt(pids);

  return 0;
}
```

3. The scheduling strategy that is most efficient in terms of overall throughput can result in an unfair distribution of total CPU time among all processes. A simple example of a high-throughput algorithm is First Come First Served (FCFS), because it always ensures that it finishes each job's burst completely. However, this can be unfair in the case where lots of short jobs arrive after a long one; the short jobs are forced to wait for the long one to finish before they can execute. A good example of a very fair algorithm is Round-Robin (RR). This

achieves fairness because it always gives each process the exact same amount of time on the CPU, distributing time completely fairly. However, throughput can suffer from unnecessary and/or inefficient context switching, as in the case where all jobs are the same length (this can be the case even if it is assumed that the process of context switching has zero overhead time).

4. A resource allocation graph shows which resources are allocated to which processes, as well as which resources processes need for completion but do not currently have. A cycle in one of these graphs represents a deadlock because it indicates a cyclic dependence among all processes, so none can acquire all the resources it needs to finish.
5. The boot loader code itself, as well as information about the specific location of the image it is supposed to load, is stored in a predefined location of the disk. When the computer is started, the BIOS executes the instructions in this sector to load the bootloader, which can then have enough information about the filesystem to continue using BIOS disk drivers to load the operating system bootstrap code.
6. start_kernel() is jumped to after all of the bootloading code has done its job, and is the first of the typical platform-independent kernel C code that really runs. It initializes various kernel subsystems and data structures. It then calls init_rest(), which spawns a kernel thread that has the kernel_init() method as an entry point. kernel_init() eventually forks and executes /sbin/init, which becomes PID 1, the first user-space program on the system. Back in init_rest(), after creating the thread that will become init, it makes the initial call to schedule(), and then creates the idle process (PID 0) and goes to sleep, through a call to cpu_idle(). As a result, this idle process is put on the CPU whenever it has no other work to do.
7. Syscall arguments are passed by register because of the barrier between user and kernel space. A user space program's stack is virtual, and managed by the kernel. So, the syscall arguments are loaded into registers and then flow jumps to the kernel syscall code and the arguments are read from those registers. Passing more than 6 arguments on Intel architecture is not supported (BUG() is called). If one wanted to pass more information, they could pass a struct via a void*
8.

```
int caught_signal = 0;

void handler (int arg)
{
  printf("Parent: signal received!\n");
  caught_signal = 1;
}


int main()
{
  // Create a shared memory space
  key_t key;
  pid_t *pids;
  int i;
  key = ftok("./370_wa1_2.c", 'a');
  if (key == -1) perror("ftok");

  i = shmget(key, 12, 0666 | IPC_CREAT); // 12 bytes = 3 pids
  if (i<0) perror("shmget");
  pids = shmat(i, NULL, 0);
  if (pids == -1) perror("shmat");
  bzero(pids,12);

  pids[0] = getpid();
  printf("getpid() : %d\n", getpid());
  printf("pids[0] : %d\n", pids[0]);
  pid_t temp;

  // Set up the signal handler
  if (signal(SIGUSR1, handler) == SIG_ERR)
    printf("Problem attaching signal handler!\n");

  pid_t child2, child3;
```

```
  // Create the first child
  if (fork() == 0)
  {
    printf("First child: sending signal to the parent\n");
    kill(pids[0], SIGUSR1);
  }

  if (getpid() == pids[0]) // Only the parent
    if ((child2 = fork()) == 0) // Second child
    {
      printf("Second child: going to sleep...\n");
      sleep(10);
      printf("Second child: Exiting!\n");
      return 0;
    }

  if (getpid() == pids[0])
  {
    if ((child3 = fork()) == 0) // Third child
    {
      printf("Third child: Will now wait...\n");
      int status;
      wait(&status);
    }
    else
    {
      printf("Parent: Waiting for signal...\n");
      while (caught_signal == 0) sleep(1);
      printf("Parent: Signal received! Waiting for second child...\n");
      int status;
      waitpid(child2, &status, 0);
      printf("Parent: Second child dead! Killing third child and exiting.\n");
      kill(child3,SIGKILL);
    }

    shmdt(pids);
  }


  return 0;
}
```