CS 241 System Programming

Discussion Section 2

30 Jan – 2 Feb

Outline

- What is a process?
- Memory layout
- Static
- fork(), wait(), exec()

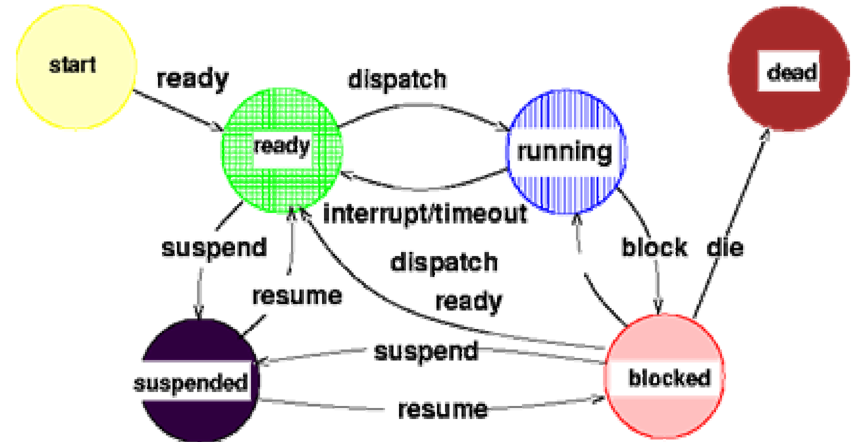
- Conditional Compilation

Program vs Process

- Program
  - .h/.c -> .o -> executable
  - Plan to complete a task
- Process
  - Instance of a program (ie currently executing)
  - Has alterable state, variables, memory

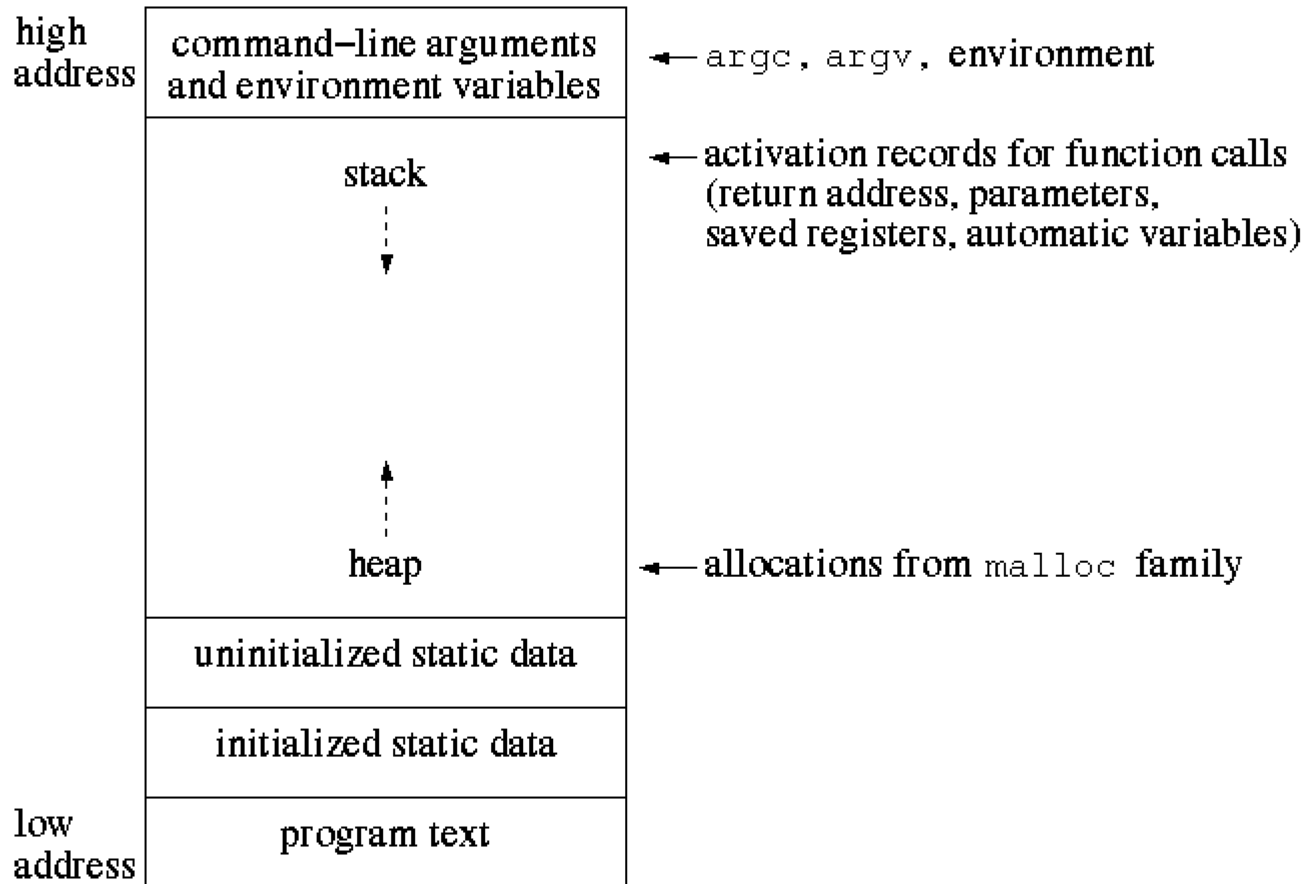
What can we do with a process?

- Execute a program
- Start more processes
- Spawn threads



- Do some kind of work.
  - File I/O (blocking)
  - Scheduling (running <-> ready)
  - Go to sleep (suspended)

Memory Layout



Memory Placement

- Variables can be stored in a variety of places:

```
int arr[400];
void func(int *b) { int a = *b; }
int main(int argc, char *argv[]) {
  int *b = malloc(sizeof(int));
  *b = 8;
  func(b);
}
```

Static Functions

- Provides for internal linkage
- Similar to “private” in C++
  - Only can be called from functions in the same file

```
static int onepass(int a[], int n);
void clearcount(void);
int getcount();
void bubblesort(int a[], int n);
```

Static Variables

```
static int count = 0;
```

- Only one instance is made per file
  - Comparable to a local variable in a function
  - All functions in file have access to it
  - No direct access from the outside
- Why would we want a static variable?

Static Example

Program 2.5

```
int a[ARRAYSIZE];
printf("Enter %d integers to sort\n", ARRAYSIZE);
for (i = 0; i < ARRAYSIZE; i++)
   scanf("%d", a+i);
printf("Array follows:\n");
for (i = 0;i < ARRAYSIZE; i++)
   printf("%2d: %4d\n", i, a[i]);
bubblesort(a, ARRAYSIZE);
printf("Sorted array follows:\n");
for (i = 0; i < ARRAYSIZE; i++)
   printf("%2d: %4d\n", i, a[i]);
printf("Number of interchanges: %d\n", getcount());
```

Static Problems

- Defined location in the executable
  - Easy to insert malicious values
- Initialized static variables waste space
  - `int arr[50000] = {1,2,3};`
    - bloats executable by 200kb

Process Creation

- Running a process from the shell
  - ls
  - uptime
- Creating a new process in C
  - pid_t fork()
    - Returns 0 to the child, and the PID of the child to the parent

Process Chain

Program 3.1

```
int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){    /* check for valid number of arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid = fork())
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child
   ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Process Fan

Program 3.2

```
int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;

   if (argc != 2){   /* check for valid number of arguments */
      fprintf(stderr, "Usage: %s processes\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if ((childpid = fork()) <= 0)
         break;

   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child 
   ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Waiting for child processes & EINTR

- Insufficient:

```
if ((childpid = fork()) > 0)
    waitpid(childpid, NULL, 0);
```

- Interrupts can happen at anytime
  - Function will return but will not have waited for child
- Better solution:

```
if ((childpid = fork()) > 0)
  while((waitpid(childpid, NULL, 0) == -1)
     && (errno = EINTR))
```

Enter restart library

- Identical functionality to system functions
  - But now they restart on interrupts
  - Download and include restart.h/restart.c
  - Functions prefixed with r_ (eg waitpid -> r_waitpid)

```
pid_t r_waitpid(pid_t pid, int *stat_loc, int opt)
{
   pid_t retval;
   while (((retval = waitpid(pid, stat_loc, opt))
       == -1) && (errno == EINTR)) ;
   return retval;
}
```

Waiting for the children to exit

Example 3.15

```
int main(int argc, char *argv[]) {
   pid_t childpid;
   int i, n;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s n\n", argv[0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if ((childpid = fork()) <= 0)
         break;

   while(r_wait(NULL) > 0) ;    /* wait for all of your children */
   fprintf(stderr, "i:%d  process ID:%ld  parent ID:%ld  child
   ID:%ld\n",
           i, (long)getpid(), (long)getppid(), (long)childpid);
   return 0;
}
```

Getting process exit status

Example 3.22

```
void show_return_status(void) {
   pid_t childpid;
   int status;

   childpid = r_wait(&status);
   if (childpid == -1)
      perror("Failed to wait for child");
   else if (WIFEXITED(status) && !WEXITSTATUS(status))
      printf("Child %ld terminated normally\n", (long)childpid);
   else if (WIFEXITED(status))
      printf("Child %ld terminated with return status %d\n",
             (long)childpid, WEXITSTATUS(status));
   else if (WIFSIGNALED(status))
      printf("Child %ld terminated due to uncaught signal %d\n",
             (long)childpid, WTERMSIG(status));
   else if (WIFSTOPPED(status))
      printf("Child %ld stopped due to signal %d\n",
             (long)childpid, WSTOPSIG(status));
}
```

Changing the process image

- Normally processes execute C code
- They can also execute other precompiled code
  - `int execv(const char *path, char *const argv[])`
    - And other variations: `execl`, `execlp`, `execv`, `execvp`
    - l takes a parameter list: argv0, argv1, argv2, …
    - v takes an array of parameters: argv[]
    - p will search for the program in the path, rather than needing to specify the whole path

Executing command-line arguments

Program 3.5

```
int main(int argc, char *argv[]) {
   pid_t childpid;
   if (argc < 2){      /* check for valid number of command-line arguments */
      fprintf (stderr, "Usage: %s command arg1 arg2 ...\n", argv[0]);
      return 1;
   }
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {                                      /* child code */
      execvp(argv[1], &argv[1]);
      perror("Child failed to execvp the command");
      return 1;
   }
   if (childpid != r_wait(NULL)) {                          /* parent code */
      perror("Parent failed to wait");
      return 1;
   }
   return 0;
}
```

Conditional Compilation

```
#ifdef DEBUG
  fprintf(stderr, “I am only printing out if
                    DEBUG is defined”);
#endif
```

- Now just compile with `–DDEBUG` when you want that information
  - Could setup with make (eg `make debug`)

Summary

- Know where your variables are being stored
  - Use this to your advantage, you can only pass around data that is in the heap or statically defined
- Spawn child processes
- Wait for child processes
- Wrap all debug statements in precompiler blocks to easily turn debug on/off