System Programming

Module 2

EGCI 252
System Programming

Computer Engineering Department
Faculty of Engineering
Mahidol University

What is a Process?

- A simple definition of processes: A process is a running program.
- The X/Open Specification defines a process as a single thread of control that executes within its address space and consumes required system resources.
- When a new process is created, a copy of the parent process' environment variables is provided as a default to the new process
- A linux's command "ps –al" shows all current processes.

Processes in Operating System

- Multi-user OS
  - Ability of an OS to have multiple users using the system concurrently or at the same time
- Multi-tasking OS
  - Ability of an OS to run multiple programs concurrently or at the same time
- Parallelism VS Concurrency
  - Execute at the same time
  - Very Fast Timesharing gives a perception of Parallelism

Processes in Operating System (cont.)

- Timesharing quantum done by the system scheduler (called swapper), which is a kernel process and has process ID of 0
- Other kernel processes are created to run the following services (various Unix kernels vary, YMMV):
  - initd (1): parent initializer of all processes
  - keventd (2): kernel event handler
  - kswapd (3): kernel memory manager
  - kreclaimd (4): reclaims pages in vm when unused
  - bdflush (5): cleans memory by flushing dirty buffers from disk cache
  - kupdated (6): maintains sanity of filesystem buffers

User and Kernel Space

- System memory is divided into two parts:
  - user space
    - a process executing in user space is executing in user mode
    - each user process is protected (isolated) from another (except for shared memory segments and mmapings in IPC)
  - kernel space
    - a process executing in kernel space is executing in kernel mode
- Kernel space is the area wherein the kernel executes
- User space is the area where a user program normally executes, except when it performs a system call.

Anatomy of a System Call

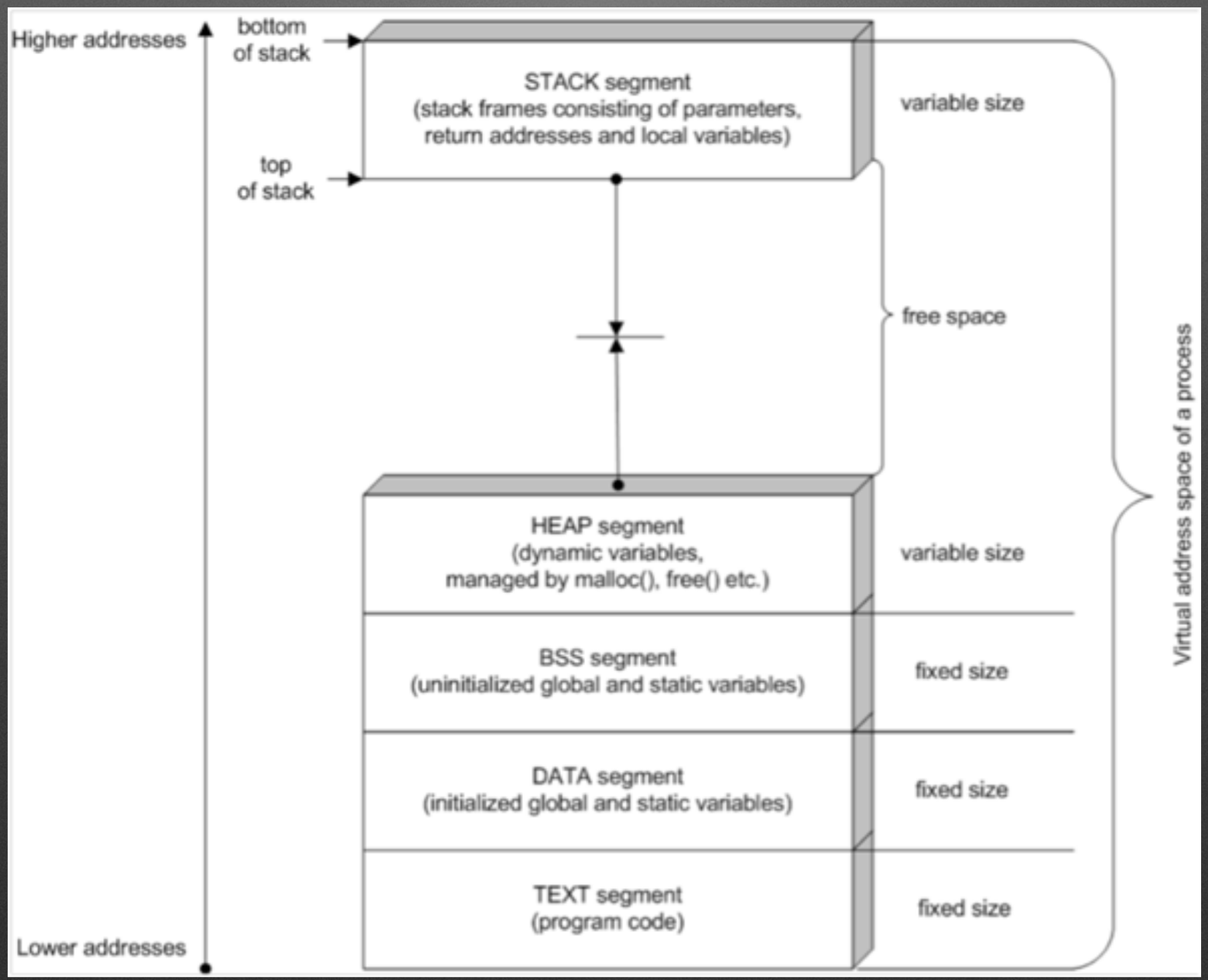
- A System Call is an explicit request to the kernel made via a software interrupt
- The standard C Library (libc) provides wrapper routines, which basically provide a user space API for all system calls, thus facilitating the context switch from user to kernel mode
- The wrapper routine (in Linux) makes an interrupt call 0x80 (vector 128 in the Interrupt Descriptor Table)
- The wrapper routine makes a call to a system call handler (sometimes called the “call gate”), which executes in kernel mode
- The system call handler in turns calls the system call interrupt service routine (ISR), which also executes in kernel mode.

Program’s Address Space

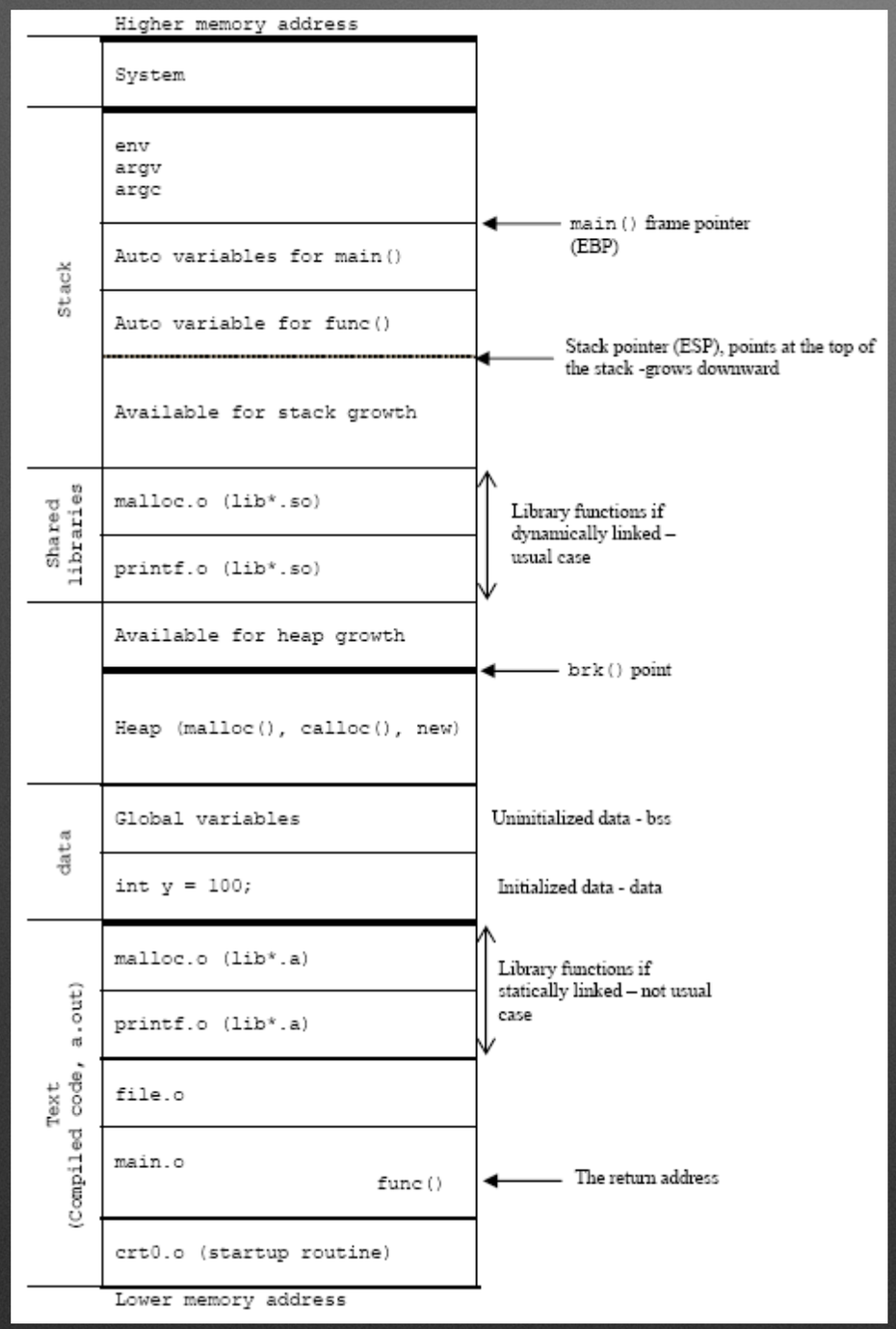
- Text Segment stores instructions of the process
- Data Segment stores
  - initialized static data : [.data]
    // char name[] = “bob”;
  - uninitialized static data : [.bss] (Block Started by Symbol)
    // int array[100];
- Heap is for dynamic memory demand // malloc();
- Stack is for function call storage and automatic variables

Program's Address Space

Picture source:
http://www.drdobbs.com/security/anatomy-of-a-stack-smashing-attack-and-h/240001832



C Language Allocation



Picture source: http://wolfie.stfu.cz

C Language Allocation (Cont.)

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

int x = 100;
char y = ‘A’;
int z;

int main(void)
{
   x = func(5);
   printf(“ %d\n”, x);
   return 0;
}
```

```
int func(int a)
{
   int *b, *c;
   b = malloc(sizeof(int));
   c = malloc(sizeof(int));
   *b = 3;
   *c = 2;
   a += x + 3;
   return a;
}
```

SEGMENT	VARIABLE : ADDRESS	DATA
Stack	0x7ffea46ab140 0x7ffea46ab138 a:0x7ffea46ab130 b:0x7ffea46ab128 c:0x7ffea46ab11c	[return address] [frame pointer] 108 0x55968fff7670 0x55968fff7690
Heap	0x55968fff7690 0x55968fff7670	2 3
Data [.bss]	z:0x55968e31c01c	0
Data [.data]	y:0x55968e31c014 x:0x55968e31c010	A 100
Text	0x55968e11a8b6 0x55968e11a73a	[main codes] [func codes]

C Language Allocation (Cont.)

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

int x = 100;
char y = ‘A’;
int z;

int func(int a)
{
  int *b, *c;

  b = malloc(sizeof(int));
  c = malloc(sizeof(int));
  *b = 3; *c = 2;

  printf("<Heap Segment>\n");
  printf(" Address of *b   = %p\n", b);
  printf(" Data of *b      = %d\n", *b);
  printf(" Address of *c   = %p\n", c);
  printf(" Data of *c      = %d\n", *c);

  a += x+3;

  printf("<Stack Segment>\n");
  printf(" Address of c    = %p\n", &c);
  printf(" Data of c       = %p\n", c);
  printf(" Address of b    = %p\n", &b);
  printf(" Data of b       = %p\n", b);
  printf(" Address of a    = %p\n", &a);
  printf(" Data of a       = %d\n", a);
  printf(" Address of RBP  = %p\n", &a + 2);
  printf(" Data of RBP     = 0x%x%08x\n”,
         *(&a + 2), *(&a + 1));
  printf(" Address of rAdd = %p\n", &a + 4);
  printf(" Data of rAdd    = 0x%x%08x\n”,
         *(&a + 4), *(&a + 3));

  while (1);

  return a;
}
```

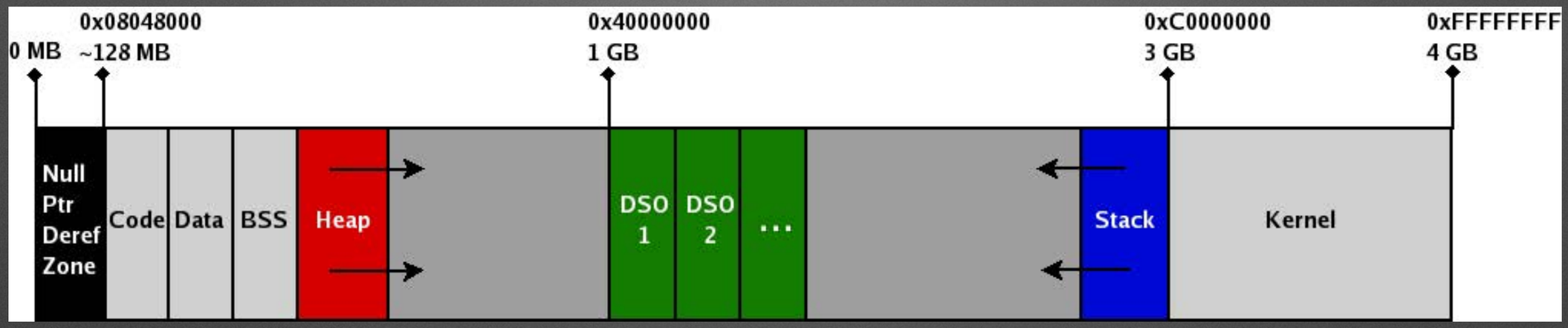
C Language Allocation (Cont.)

```
int main(void)
{
   printf("<Text Segment>\n");
   printf(" Address of main = %p\n", main);
   printf(" Address of func = %p\n", func);

   printf("<Data Segment>\n");
   printf(" <<Initialized Data>>\n");
   printf("  Address of x   = %p\n", &x);
   printf("  Data of x      = %d\n", x);
   printf("  Address of y   = %p\n", &y);
   printf("  Data of y      = %c\n", y);
   printf(" <<Uninitialized Data>>\n");
   printf("  Address of z   = %p\n", &z);
   printf("  Data of z      = %d\n", z);

   x = func(5);
   printf("[x = %d]\n", x);

   return 0;
}
```

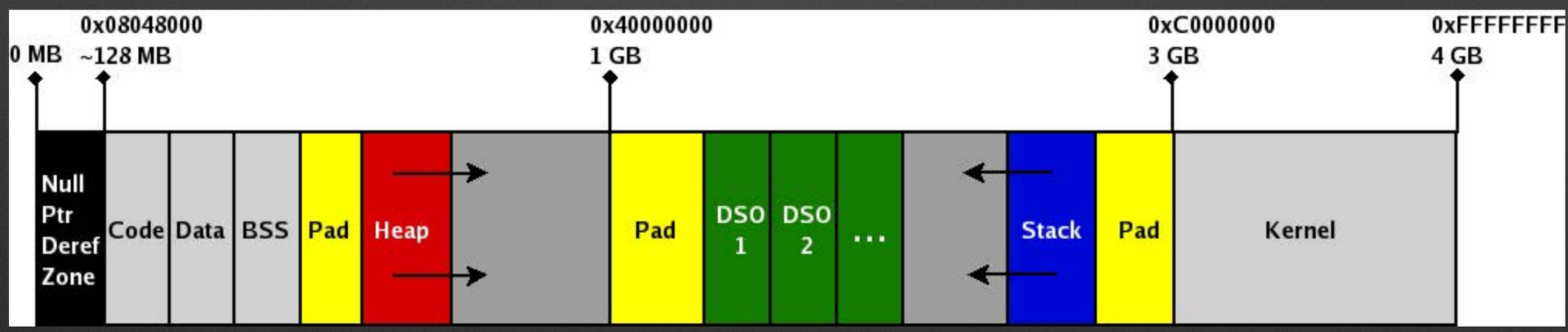
```
//Compile with Clang Only
//On both MacOS and Linux
$clang -o proc_space proc_space.c
$cat /proc/<pid>/maps
```

Program's Address Space in Linux

Conventional Address Space Layout



Address Space Layout with Red Hat Exec-Shield



The Linux Process Descriptor

- Each Linux process is described by a task_struct structure defined in include/linux/sched.h
- This structure holds information on most aspects of a process in memory, including, among other items:
  - process state
  - next and previous task pointers
  - next and previous runnable task pointers
  - Parent, Child, and Sibling pointers
  - tty information
  - current directory information
  - open file descriptors table
  - memory pointers
  - signals received

Task State

- TASK_RUNNING: running or waiting to be executed
- TASK_INTERRUPTIBLE: a sleeping or suspended process, can be awakened by signal
- TASK_STOPPED: process is stopped (as by a debugger or SIGTSTP, Ctrl-Z)
- TASK_ZOMBIE: process is in “walking dead” state waiting for parent process to issue wait() call
- TASK_UNINTERRUPTIBLE: task is performing critical operation and should not be interrupted by a signal (usually used with device drivers)

Process Attributes

- Process ID:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

  - Every unix process has an associated process id (pid)
  - each new process is assigned a new unique unused pid
  - The pid on the 64-bit system usually ranges from 0 to 4194304
  - pids roll over after 4194304 and assignment begins again at 0, issuing unused pids

Process Ids and init

- Every process on the system has a parent, with the exception of pid 1: init
  - the init process “hangs around”, it is responsible for the initialization and booting of the system, and for running any new programs, like the login program, and your shell
  - Init executes /etc/rc* files during initialization, and is the ultimate parent of every subsequent process in the system
  - If init is killed, the system shuts down
- A process’s parent id (ppid) can be obtained with the pid_t getppid(void) call.

Environments

- All processes by default inherit the environment of their parent process
- The environment can be obtained through the char * environ[] variable.
- char * getenv(const char * name) will return the associated value for the name passed in:
  - char * path = getenv(“PATH”);
- int setenv(const char * name, const char * value, int overwrite) will set an environment variable
- examples: environ.c

fork()

- fork() creates a new child process
- the OS copies the current program into the new process, resets the program pointer to the start of the new program (child fork location), and both processes continue execution independently as two separate processes
- The child gets its own copy of the parent's:
  - data segments
  - heap segment
  - stack segment
  - file descriptors

fork() Return Values

- fork() is the one Unix function that is called once but returns twice:
- If fork() returns 0:
  - you’re in the new child process
- If fork() returns > 1 (i.e., the pid of the new child process)
  - you’re back in the parent process

fork( ) Example

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


int main()
{
     pid_t pid; char *msg; int n;

     printf("fork program starting\n");
     pid = fork();
     switch(pid)
     {
          case -1: _exit(1);
          case 0:  msg = "Child!\n"; n = 45; break;
          default:  msg = "Parent!\n"; n = 3; break;
     }


     for (; n>0; n--) { puts(msg); sleep(1); }
     _exit(0);
}
```

Waiting on Our Children

- Unlike life, parents should always hang around for their children's lives (runtimes) to end, that is to say:
  - Parent processes should always wait for their child processes to end
- When a child process dies, a SIGCHLD signal is sent to the parent as notification
- The SIGCHLD signal's default disposition is to ignore the signal
- A parent can find out the exit status of a child process by calling one of the wait() functions

Waiting on Children (Cont.)

- Parent processes find out the exit status of their children by executing a wait() call:
  - pid_t wait(int * status);
  - pid_t waitpid(pid_t pid, int * status, int options);
- Wait() blocks until it receives the exit status from a child
- Waitpid can wait on a specific child, and doesn't necessarily block (WNOHANG)
- Waiting allows the parent to obtain the return value from the child's process

waitpid()

pid_t waitpid(pid_t pid, int * status, int options);

- pid can be any of 4 values:
  - < -1: wait for any child whose gpid is the same as |pid|
  - == -1: waits for any child to terminate
  - == 0: waits for a child in the same process group as the current process
  - > 0: waits for the specified process pid to exit
- The following macros work on status:
  - WIFEXITED(status): true if process exited normally
  - WIFSIGNALED(status): true if process was killed by a signal

wait( ) Example

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>


int main()
{
    pid_t pid; char *msg; int n;

printf("fork program starting\n");
pid = fork();
switch(pid)
{
      case -1: _exit(1);
      case 0:  msg = "Child!\n"; n = 45; break;
      default:  msg = "Parent!\n"; n = 3; break;
}


for (; n>0; n--) { puts(msg); sleep(1); }
```

```
if (pid)
{
      int stat_val; pid_t child_pid;

      child_pid = wait(&stat_val);
      printf("Child has finished: PID = %d\n",
              child_pid);
       if (WIFEXITED(stat_val))
           printf("Child exited with code %d\n",
                   WEXITSTATUS(stat_val));
           else
               printf("Child terminated abnormally\n");
                }

                _exit(0);
            }
```

Problem Children: Orphans and Zombies

- If a child process exits before it's parent has called wait(), it would be inefficient to keep the entire child process around, since all the parent is going to want to know about is the exit status:
  - A zombie is a child process that that has exited before it's parent's has called wait() for the child's exit status
  - A zombie holds nothing but the child's exit status (held in the program control block)
  - Modern Unix systems have init (pid == 1) adopt zombies after their parents die, so that zombies do not hang around forever as they used to, in case the parent never did get around to calling wait

Orphans and Zombies

- If a parent process dies before it's child, the child process becomes an orphan
- An orphan is a child process whose parent is no longer living
- An orphan is immediately "adopted" by the init process (pid == 1), who will call wait() on behalf of the deceased parent when the child dies

Zombie Process : Example

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid; char *msg; int n;

printf("fork program starting\n");
pid = fork();
switch(pid)
{
      case -1: _exit(1);
      case 0:  msg = "Child!\n"; n = 3; break;
      default:  msg = "Parent!\n"; n = 45; break;
}

for (; n>0; n--) { puts(msg); sleep(1); }
```

```
if (pid)
{
      int stat_val; pid_t child_pid;

      child_pid = wait(&stat_val);
      printf("Child has finished: PID = %d\n",
            child_pid);
       if (WIFEXITED(stat_val)
           printf("Child exited with code %d\n",
                  WEXITSTATUS(stat_val);
      else
           printf("Child terminated abnormally\n");
}

_exit(0);
}
```

Note: Linux systems will usually refer a zombie as a defunct.

Using the exec() Family

- The exec functions replace the program running in a process with another program. When a program calls an exec function, that process immediately ceases executing that program and begins executing a new program from the beginning, assuming that the exec call doesn't encounter an error.
- What happens with "exec" ?

- Before "fork( )"

```
int main() {
  …
  int pid = fork( );
  if (pid == 0) exec( … )
```

- After "fork( )"

```
int main() {
  …
  int pid = fork( );
  if (pid == 0) exec( … )
```

```
int main() {
  …
  int pid = fork( );
  if (pid == 0) exec( … )
```

- After "exec( )"
  - One process still contains the original code we started with
  - Other process now has completely different code

```
int main() {
  …
  int pid = fork( );
  if (pid == 0) exec( … )
```

```
int main() {
  // another program
  // starts at beginning
}
```

The exec() Functions: Out with the old, In with the new

- The exec() functions all replace the current program running within the process with another program
- bring up an xterm:
  - exec sleep 5 #what happens and why?
- There are two families of exec() functions, the “l” family (list), and the “v” family (vector)
- Each exec() call can choose different ways of finding the executable and whether the environment is delivered in the form of a list or an array (vector)
- The environment, open file handles, etc. are passed into the exec’d program
- What is the return value of an exec() call?

The execl... functions

- int execl(const char * path, const char * arg0, ...);
  - executes the command at path, passing it the environment as a list: arg0 ... argn
  - thus, the execl family breaks down argv into its individual constituents, and then passes them as a list to the execl? function (the l stands for list)
- int execlp(const char * path, const char * arg0, ...);
  - same as execl, but uses $PATH resolution for locating the program in path, thus an absolute pathname is not necessary
- int execle(const char * path, const char * arg0, ... char * envp[]);
  - allows you to specifically set the new program's environment, which replaces the default current program's environment

The execv... functions

- int execv(const char * path, char *const argv[]);
  - executes the command at path, passing it the environment contained in a single argv[] vector
- int execvp(const char * path, char *const argv[]); same as execv, but uses $PATH resolution for locating the program in path
- int execve(const char * path, char *const argv[], char * const envp[]);

Using fork and exec Together

- A common pattern to run a subprogram within a program is first to fork the process and then exec the subprogram. This allows the calling program to continue execution in the parent process while the calling program is replaced by the subprogram in the child process.

Execl Example

- print.c

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

int main(int argc, char* argv[ ])
{
   int num = atoi(argv[1]);
   int loops = atoi(argv[2]);
   int a;

   setbuf(stdout, NULL);
   for(a = 0; a < loops; a++) { printf(“%d”, num); sleep(1); }
   printf(“\n”)
   return 0;
}
```

Excel Example (Cont.)

- execl_demo.c

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

int main()
{
   pid_t pid = fork();
   printf(“Process ID: %d\n”, (int) pid);
   if (pid == 0)
         execl(“./print”, “print”, “1”, “10”, NULL);
   else
         execl(“./print”, “print”, “2”, “20”, NULL);
}
```

Execv Example

- execv_demo.c

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

int main()
{
   char* argv1[] = {“print”, “1”, “10”, NULL};
   char* argv2[] = {“print”, “2”, “20”, NULL};
   pid_t pid = fork();

   printf(“Process ID: %d\n”, pid);
   if(pid == 0)
         execv(“./print”, argv1);
   else
         execv(“./print”, argv2);
}
```

vfork() and Copy On Write

- When a process forks, the entire current process (plus segments, environment, etc.) is copied over to the new process
- When that new process called exec(), the entire address space is replaced (overlaid) with the new environment of the exec'ing program
- Efficiency question: If you know you're going to call exec immediately after fork, why have fork spend time copying the entire address space over when we know it's just going to get overwritten immediately on the exec() call?
- Answer: vfork() doesn't copy entire address space

Sessions and Process Groups

- A process group is a group of related processes, that share some common interest, as all the processes in a pipeline do:
  - ls –l | sort | wc –l
- A session is a further abstracted group of related process groups or individual processes, such as all the jobs in a given terminal shell session
- Sessions are generally created during login, and process groups are managed by the job processing capabilities of a given shell

Priorities and Being Nice

- The scheduler recognizes processes of three different scheduling policies:
  - SCHED_FIFO (unalterable real-time processes)
  - SCHED_RR (alterable real-time processes)
  - SCHED_OTHER (conventional, time-shared)
- Processes with SCHED_OTHER policy are assigned a default dynamic priority of 0, and can voluntarily lower their priority by incrementally raising their “niceness” value, up to 10 (range is -20 to +19, effectively 1 – 140 in terms of process priority)

Debugging Multiple Processes

- Debugging processes that fork can be a little tricky, because whereas once you had one process, now you have two.
- Which process will gdb debug? Answer: the parent
- How do you debug the child process?
- With another gdb (ddd) session:
- Add a sleep() call at the start of the child code
- run gdb (ddd) on the program, and set a breakpoint right after the sleep() call in the child section
- run the first gdb session on the parent
- after the fork(), attach to the child process and then issue the “continue” call in the child gdb session

Death and Destruction

- All processes usually end at some time during runtime (with the exception of init)
- Processes may end either by:
  - executing a return from the main function
  - calling the exit(int) function
  - calling the _exit(int) function
  - calling the abort(void) function
  - generates SIGABRT signal, core dumps and then exits
- When a process exits, the OS delivers a termination status to the parent process of the recently deceased process

End of Module