Week 6.1

Processes

School of Information Technology and Electrical Engineering
The University of Queensland

References

Credits:

- Glass & Ables (pages 472 to 489, 584 to 590, 600 to 606)
- Bryant and O'Halloran, "Computer Systems: A Programmer's Perspective"
- Silberschatz et. al, "Operating Systems concepts"
- Tanenbaum, "Modern Operating Systems"
- Rochkind, "Advanced UNIX Programming"

Recall from Week Three

Operating Systems provide *abstractions* to

- make computer hardware easier to use
  - for user, programmer, system administrator…
- manage hardware resources

Example abstractions

Abstraction	**Resource**
Virtual Memory	Memory
Processes	CPU time+
Files	Disk space

Processes

Definition: A process is an instance of a running program

- One of the most profound ideas in computer science
- Not the same as "program" or "processor"

Process provides each program with key abstractions:

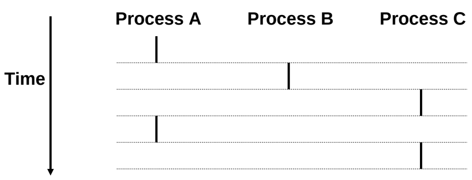
- Logical control flow
  - Each program seems to have exclusive use of the CPU
- Private address space
  - Each program seems to have exclusive use of main memory
  - A collection of system resources.

How are these illusions maintained?

- Process executions interleaved (multitasking)
- Address spaces managed by virtual memory system
  - Each process has its own address space.

Logical Control Flows

Each process has its own logical control flow



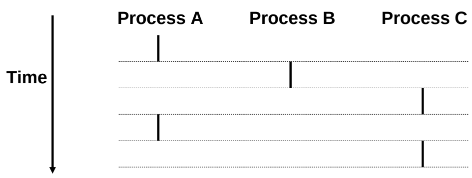
Concurrent Processes

Two processes ***run concurrently*** (*are* ***concurrent***) if their flows overlap in time

Otherwise, they are ***sequential***

Examples:

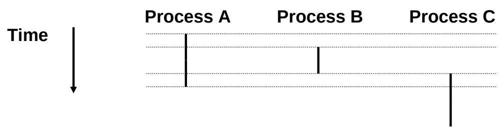
- Concurrent: A & B, A & C
- Sequential: B & C



User View of Concurrent Processes

Control flows for concurrent processes are physically disjoint in time

However, user can think of concurrent processes as running in parallel with each other



`fork`: Creating new processes

`int fork(void)`

- creates a new process (child process)
  - child is identical to calling process (parent process)
- returns 0 to the child process
- returns child's `pid` to the parent process
  - `pid` = process-id (numerical id)

```
pid_t pid=fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because when it is called, there is one process, when it returns, there are two

Fork Example #1

Key Points

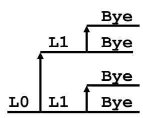
- Parent and child both run same code
  - Distinguish parent from child by return value from `fork`
- Start with same state, but each has private copy
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork_one()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
       printf("Child has x = %d\n", ++x);
    } else {
       printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```


Fork Example #2

Both parent and child can continue forking

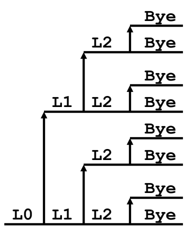
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

As many times as they like…

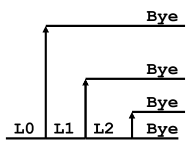
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

Common for parent to be the only one to continue forking

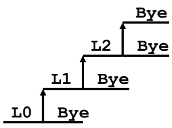
```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
          printf("L2\n");
          fork();
       }
    }
    printf("Bye\n");
}
```



Fork Example #5

But it could be just the child…

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Break

`exit`: Destroying Process

`void exit(int status)`

- exits a process
  - Status 0 for no error.

`atexit(void (*func)(void))`

- registers functions to be executed upon exit
- Argument to `atexit()` is a function pointer
- Functions called in reverse order of registration
  - Functions not called if process exits abnormally

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```


`exit` (continued)

`exit()` – buffers are flushed, files are closed

`return` from `main()` is the same as `exit()` at that point

- Any registered exit functions will be executed

`_exit()` [Note the underscore]

- Will not run the registered functions
- May not flush buffers
- Can be used for child processes, so as to not interfere with parent

Zombies

Idea

- When process terminates, still consumes system resources
  - Various tables maintained by OS
- Called a "zombie"
  - Living corpse, half alive and half dead

Reaping

- Performed by parent on terminated child
- Parent is given exit status information
- Kernel discards process

What if Parent Doesn't Reap?

- If any parent terminates without reaping a child, then child will be adopted by & later reaped by `init` process
- Only need explicit reaping for long-running processes
  - E.g., shells and servers

Zombie Example

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

```
=> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
=> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
=> kill 6639
[1]    Terminated
=> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

`ps` shows child process as "defunct"

Killing parent allows child to be reaped

Nonterminating Child Example

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

```
=> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
=> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
=> kill 6676
=> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

Child process still active even though parent has terminated

Must kill explicitly, or else will keep running indefinitely

wait: Synchronizing with children

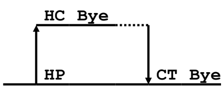
`int wait(int *child_status)`

- suspends current process until one of its children terminates
- return value is the `pid` of the child process that terminated
- if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

wait: Synchronizing with children (cont.)

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit(0);
}
```



Wait Example

If multiple children completed, will take in arbitrary order

Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```


Waitpid

`waitpid(pid, &status, options)`

- Can wait for specific process
- Various options

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
```


Wait/Waitpid Example Outputs

Using `wait` (fork10)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using `waitpid` (fork11)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```

exec: Running new programs

`int execl(char *path, char *arg0, char *arg1, …, 0)`

- loads and runs executable at `path` with args `arg0`, `arg1`, …
  - `path` is the complete path of an executable
  - `arg0` becomes the name of the process
    - typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`
  - "real" arguments to the executable start with `arg1`, etc.
  - list of args is terminated by a `(char *)0` argument
- returns `-1` if error, otherwise doesn't return!

```
main() {
   if (fork() == 0) {
      execl("/usr/bin/ls", "ls", "-al", 0);
   }
   wait(NULL);
   printf("Listing completed\n");
   exit(0);
}
```


exec variations

`execl()`	`execve()`
`execv()`	`execlp()`
`execle()`	`execvp()`

`l` – arguments directly in call (list)
`v` – arguments in array (vector)
`p` – use PATH to find program
- Otherwise provide full path of program

`e` – provide environment definition

See `exec` man page for details (man –s2 exec on agave)

System Call Summary

Basic Functions
- `fork()` spawns new process
  - Called once, returns in two processes
- `exit()` terminates own process
  - Called once, never returns
  - Puts it into "zombie" status
- `wait()` and `waitpid()` wait for and reap terminated children
- `execl()` and variants run a new program in an existing process
  - Called once, (normally) never returns

Programming with Processes

Programming Challenges
- Understanding the nonstandard semantics of the functions
- Avoiding improper use of system resources
  - E.g. "Fork bombs" can disable a system.

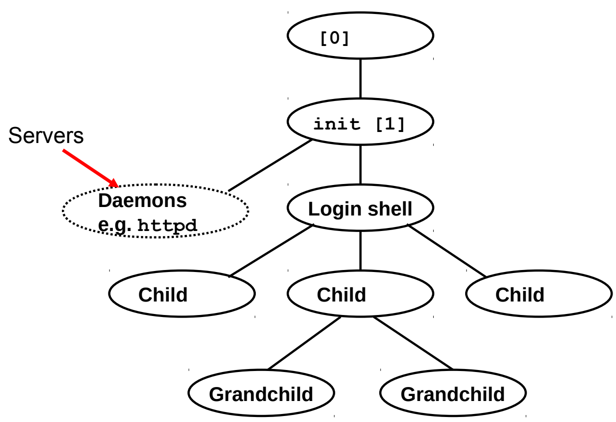
Resource sharing options
- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

Execution options
- Parent and children execute concurrently
- Parent waits until children terminate

Useful UNIX

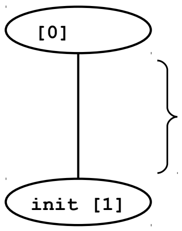
ps
top
kill
xargs

Unix Process Hierarchy



Unix Startup: Step 1

1. On reset, CPU starts a small bootstrap program
2. Bootstrap program loads the boot block (disk block 0)
3. Boot block program loads kernel binary (e.g., /boot/vmlinux)
4. Boot block program passes control to kernel
5. Kernel handcrafts the data structures for process 0

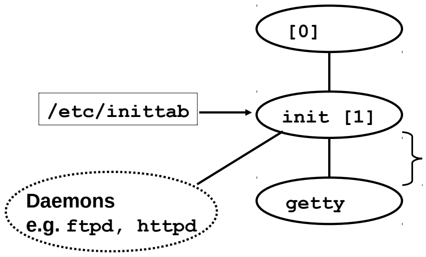


Process 0: handcrafted kernel process

Process 0 forks child process 1

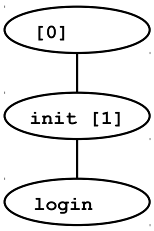
Child process 1 execs `/sbin/init`

Unix Startup: Step 2



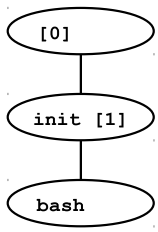
`init` forks and execs daemons per `/etc/inittab`, and forks and execs a `getty` program for the console

Unix Startup: Step 3



The `getty` process execs a `login` program

Unix Startup: Step 4



`login` reads login and passwd.
if OK, it execs a *shell*.
if not OK, it execs another `getty`

Shell Programs

A *shell* is an application program that runs programs on behalf of the user.

- `sh` – Original Unix Bourne Shell
- `csh` – BSD Unix C Shell, `tcsh` – Enhanced C Shell
- `bash` –Bourne-Again Shell

```
int main()
{
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);
        /* evaluate */
        eval(cmdline);
    }
}
```

- Execution is a sequence of read/evaluate steps

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) {    /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else          /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

Problem with Simple Shell Example

Shell correctly waits for and reaps foreground jobs

But what about background jobs?

- Will become zombies when they terminate
- Will never be reaped because shell (typically) will not terminate
- Creates a memory leak that will eventually crash the kernel when it runs out of memory

Solution: Reaping background jobs requires a mechanism called a *signal*

Signals

A *signal* is a small message that notifies a process that an event of some type has occurred in the system

- Kernel abstraction for exceptions and interrupts
- Sent from the kernel (sometimes at the request of another process) to a process
- Different signals are identified by small integer IDs
- Only information in a signal is its ID and the fact that it arrived
- More on signals later (inter-process communication)

ID	Name	Default Action	Corresponding Event
2	`SIGINT`	Terminate	Interrupt from keyboard (`Ctrl-c`)
9	`SIGKILL`	Terminate	Kill program (cannot override or ignore)
11	`SIGSEGV`	Terminate	Segmentation violation
14	`SIGALRM`	Terminate	Timer signal
17	`SIGCHLD`	Ignore	Child stopped or terminated

Signal Handling

An application program can specify a function called a signal handler to be invoked when a specific signal is received. A process can deal with a signal in one of the following ways:

- The process can let the default action happen
- The process can block the signal (some signals cannot be ignored)
- the process can catch the signal with a handler.

To establish a signal handler:

- `#include <signal.h>`
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - To see what is in struct sigaction look at the man page.
- There is an older function with a simpler interface called `signal` but it isn't as predictable.

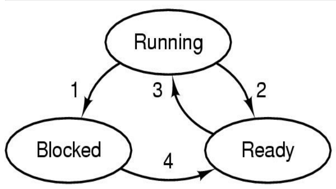
Signal Handling

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

void dostuff(int s) {
    fprintf(stderr, "Got signal %d\n",s);
}

int main(int argc, char** argv) {
    struct sigaction sa;
    sa.sa_handler=dostuff;
    sa.sa_flags=SA_RESTART;  // restart syscalls if interrupted
    sigaction(SIGINT, &sa, 0);
    while (1){sleep(10);}
}
```


Process States (1)



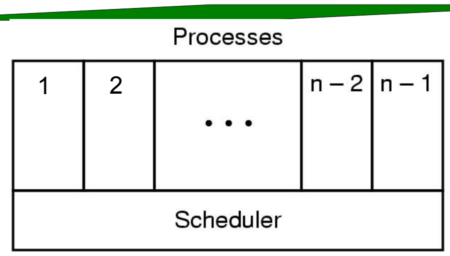
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Possible process states

- running
- blocked
- ready

Transitions between states shown

Process States (2)



Lowest layer of process-structured OS

- handles interrupts, scheduling

Above that layer are sequential processes

Implementation of Processes

Fields of a **process control block**

- Data structure - one per process
- Contains information about the process

Process management	Memory management	File management
Registers Program counter Program status word Stack pointer Process state Priority Scheduling parameters Process ID Parent process Process group Signals Time when process started CPU time used Children's CPU time Time of next alarm	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Context Switch

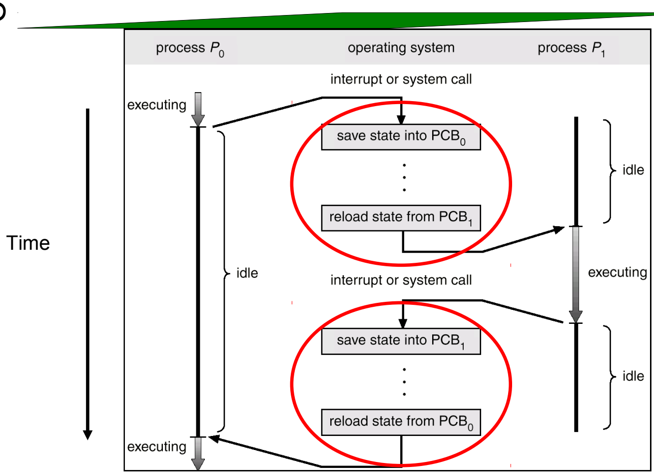
When CPU switches to another process, the system must

- save the state of the old process; and
- load the saved state for the new process

Context-switch time is overhead

- system does no useful work while switching
- Time is dependent on hardware support

Context Switch (cont.)



Process Scheduling Queues

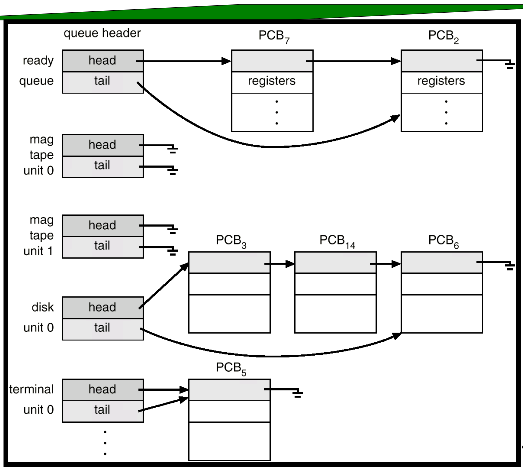
Job queue – set of all processes in the system

Ready queue – set of all processes residing in main memory, ready and waiting to execute

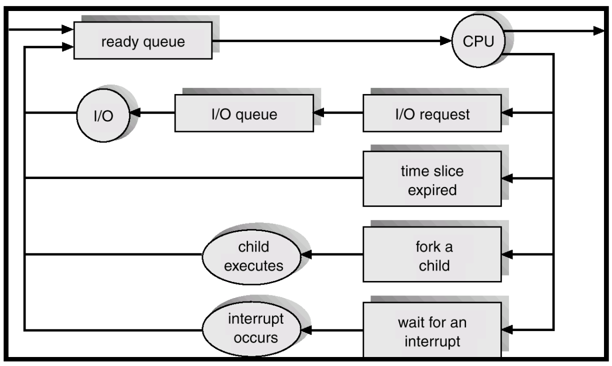
Device queues – set of processes waiting for an I/O device

Processes migrate between the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling



CSSE 2310 7231

Schedulers

Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue

Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU

CSSE 2310 7231

Schedulers (Cont.)

Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)

Long-term scheduler is invoked very infrequently (seconds) ⇒ (may be slow)

The long-term scheduler controls the *degree of multiprogramming*

Processes can be described as either:

- I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts
- *CPU-bound process* – spends more time doing computations; few very long CPU bursts

Scheduling algorithms are a topic for the Operating Systems Architecture course (COMP3301)

CSSE 2310 7231

Reminder…

Assignment Two – due tonight

Mid-semester exam this Friday