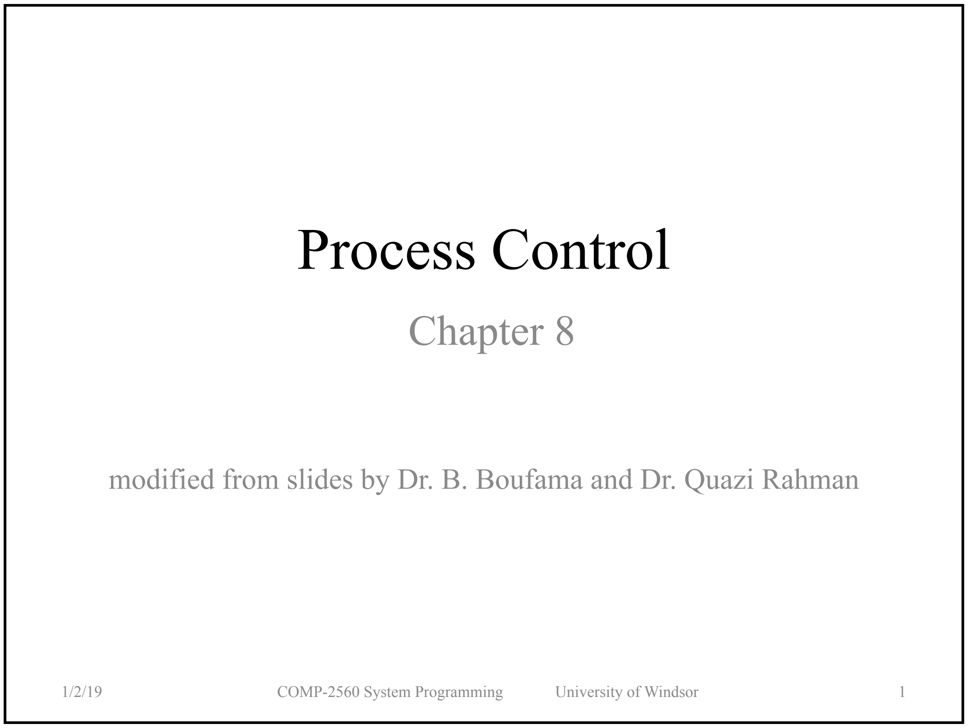
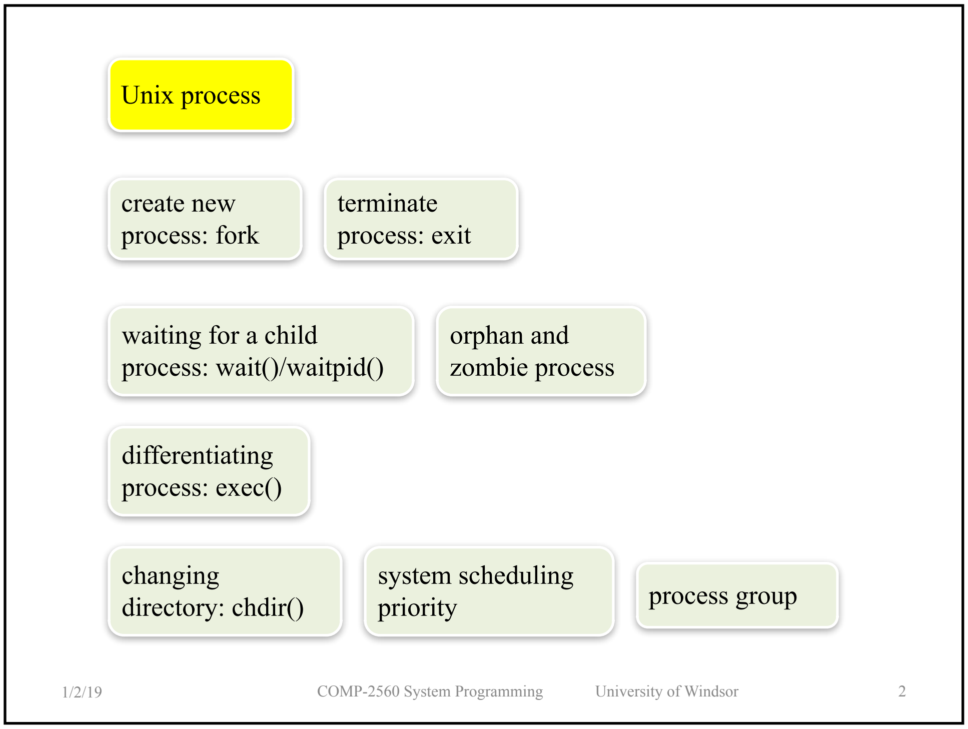


Process Control

Chapter 8

modified from slides by Dr. B. Boufama and Dr. Quazi Rahman



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

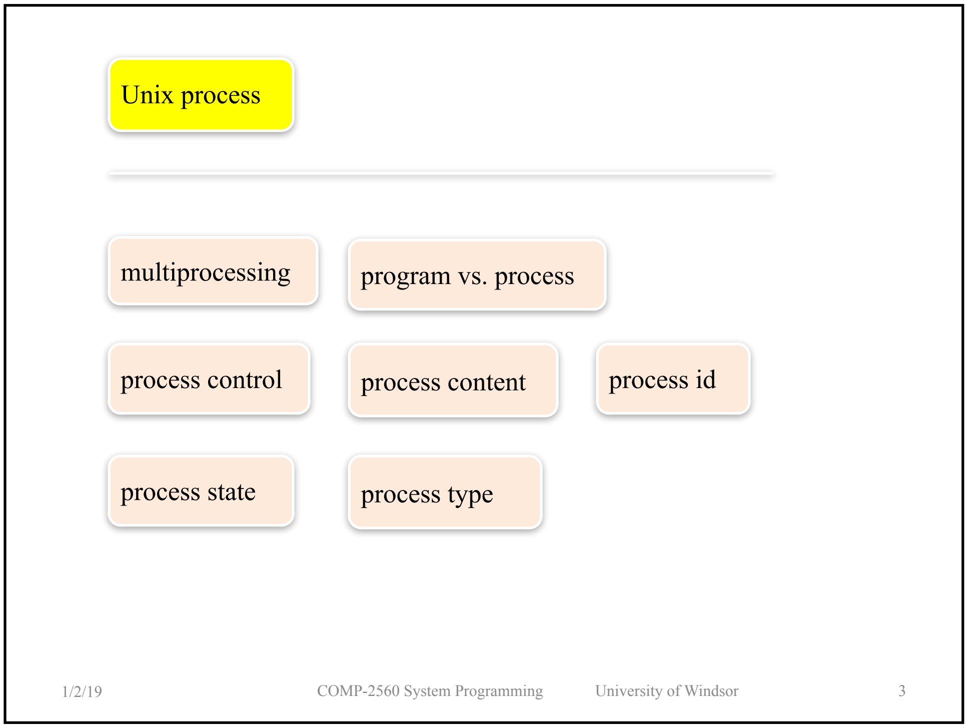
orphan and zombie process

differentiating process: exec()

changing directory: chdir()

system scheduling priority

process group



Unix process

multiprocessing

program vs. process

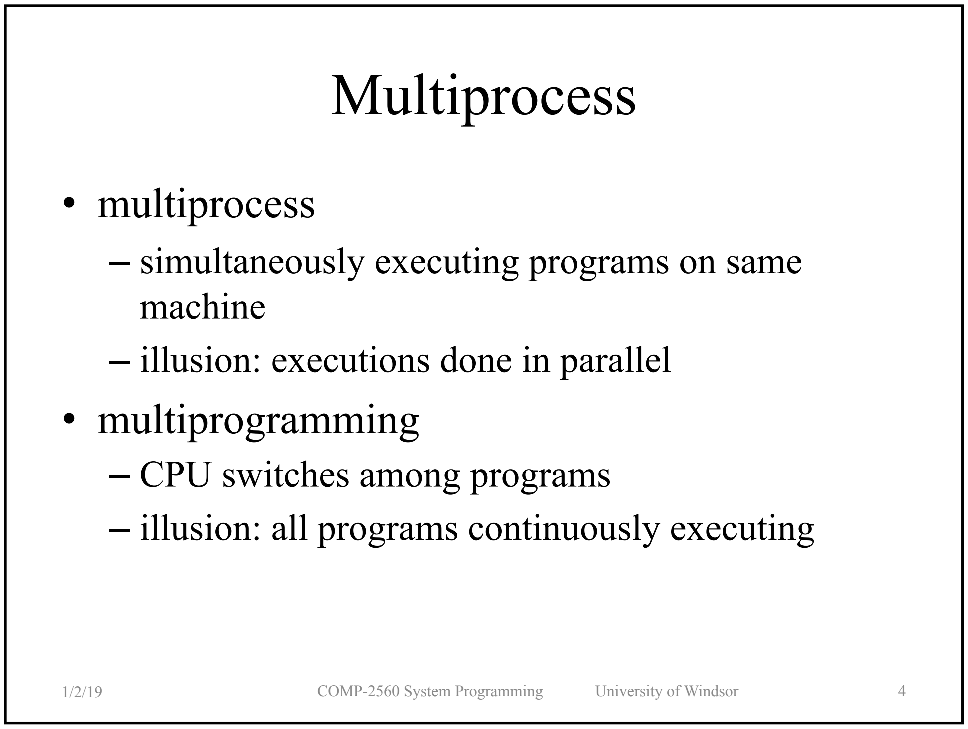
process control

process content

process id

process state

process type



Multiprocess

- multiprocess
  - simultaneously executing programs on same machine
  - illusion: executions done in parallel
- multiprogramming
  - CPU switches among programs
  - illusion: all programs continuously executing

Program vs. Process

- program
  - (an executable) file (residing on a disk)
- process
  - executing (running) program
  - opened in the working memory (RAM)
  - usually with a limited life-time
  - also called *task*
- a running program -> process

Process Control

- what is included in process control
  - creation of new processes
  - program execution
  - process termination
- what is included in a process
  - process ID
  - user ID, group ID

What Does a Unix Process Contain ?

- a unique process ID
- user ID of the owner
- code segment
- data segment (variables)
- stack segment
- an environment

- nonnegative
- assigned by OS
- used to identify a process

instructions that are being executed

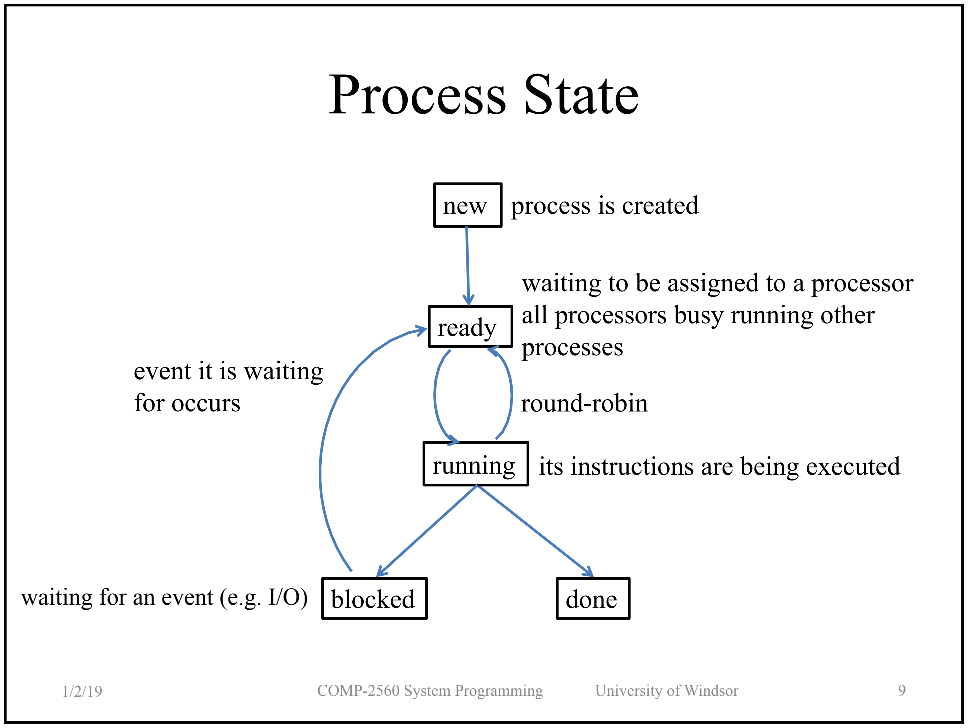
a form of memory where it is possible to push and pop instructions

e.g.
registers' contents
tables of open files

Process ID

- system calls getpid() and getppid()

```
#include <stdio.h>

int main(void) {
    printf("Hello, my PID is %d\n", getpid());
    printf("Hello, my PPID is %d\n", getppid());
    exit(0);
}

> a.out
Hello, my PID is 11723
Hello, my PPID is 5598
```



Process State

new

process is created

ready

waiting to be assigned to a processor
all processors busy running other processes

round-robin

running

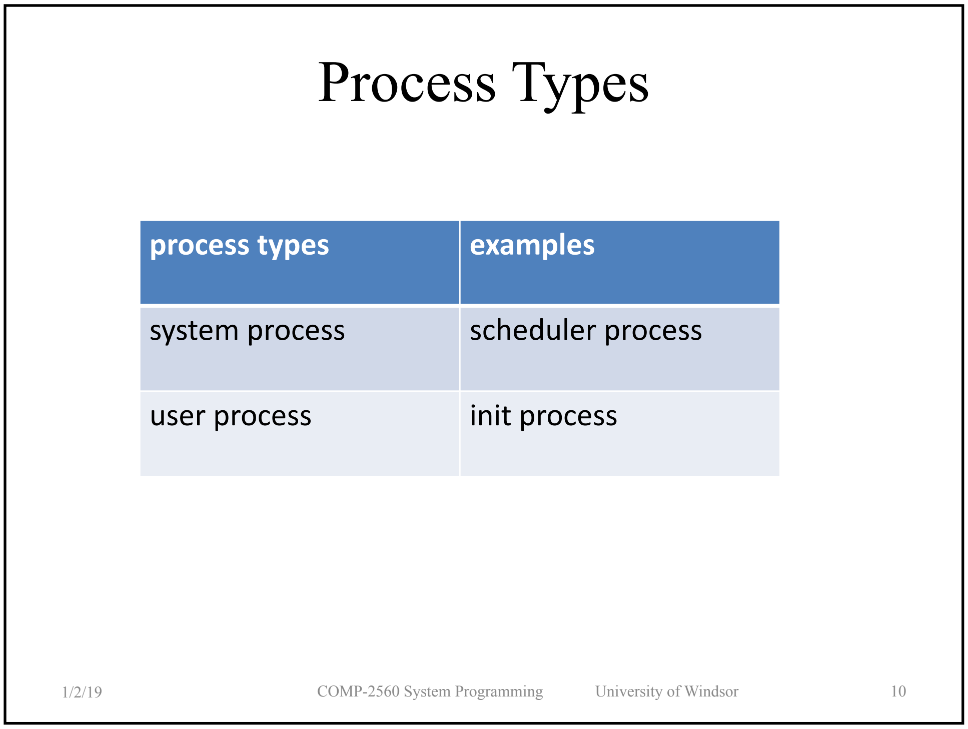
its instructions are being executed

event it is waiting for occurs

waiting for an event (e.g. I/O)

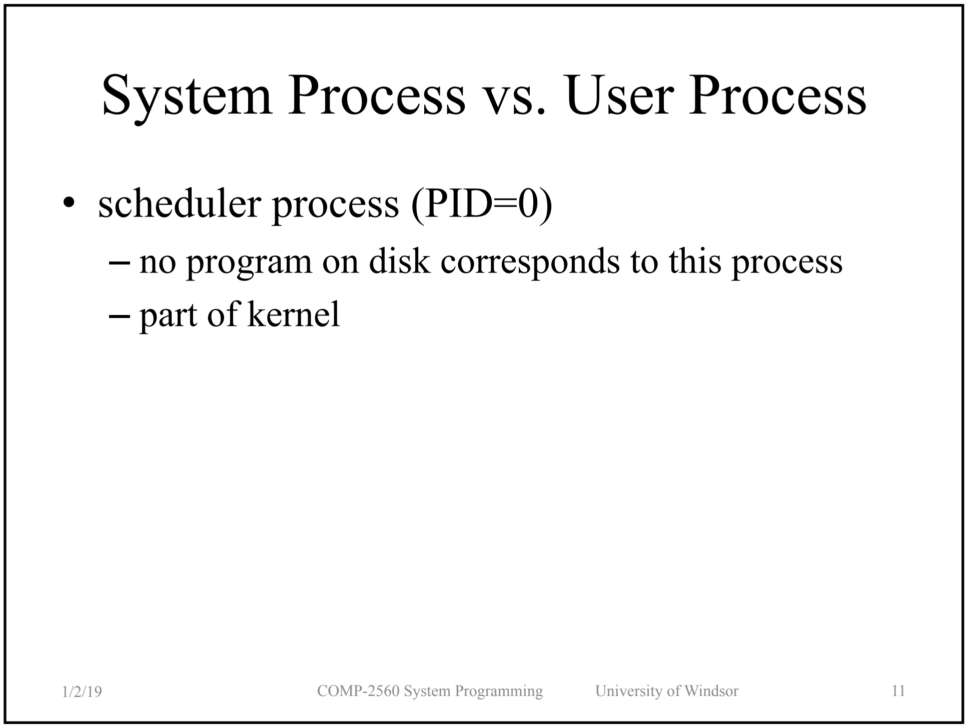
blocked

done



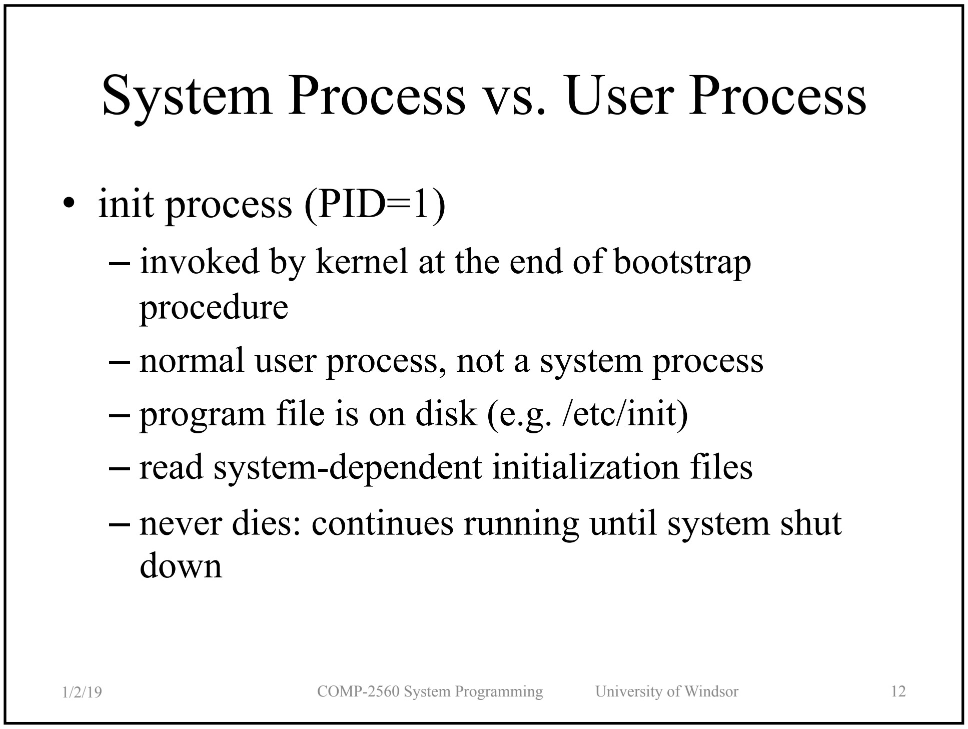
Process Types

process types	examples
system process	scheduler process
user process	init process



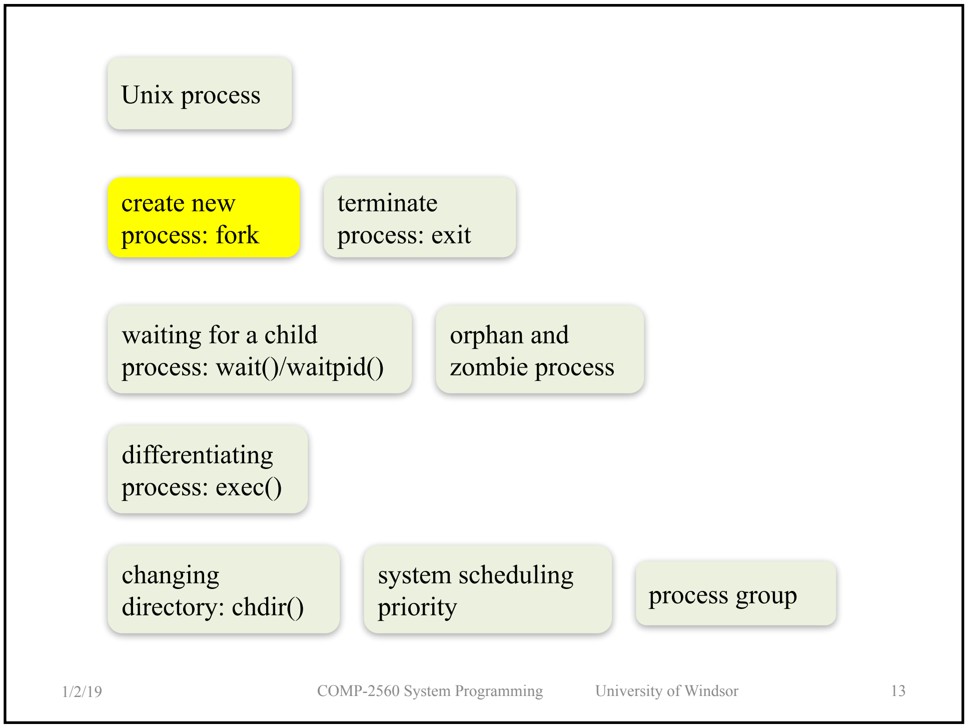
System Process vs. User Process

- scheduler process (PID=0)
  - no program on disk corresponds to this process
  - part of kernel



System Process vs. User Process

- init process (PID=1)
  - invoked by kernel at the end of bootstrap procedure
  - normal user process, not a system process
  - program file is on disk (e.g. /etc/init)
  - read system-dependent initialization files
  - never dies: continues running until system shut down



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

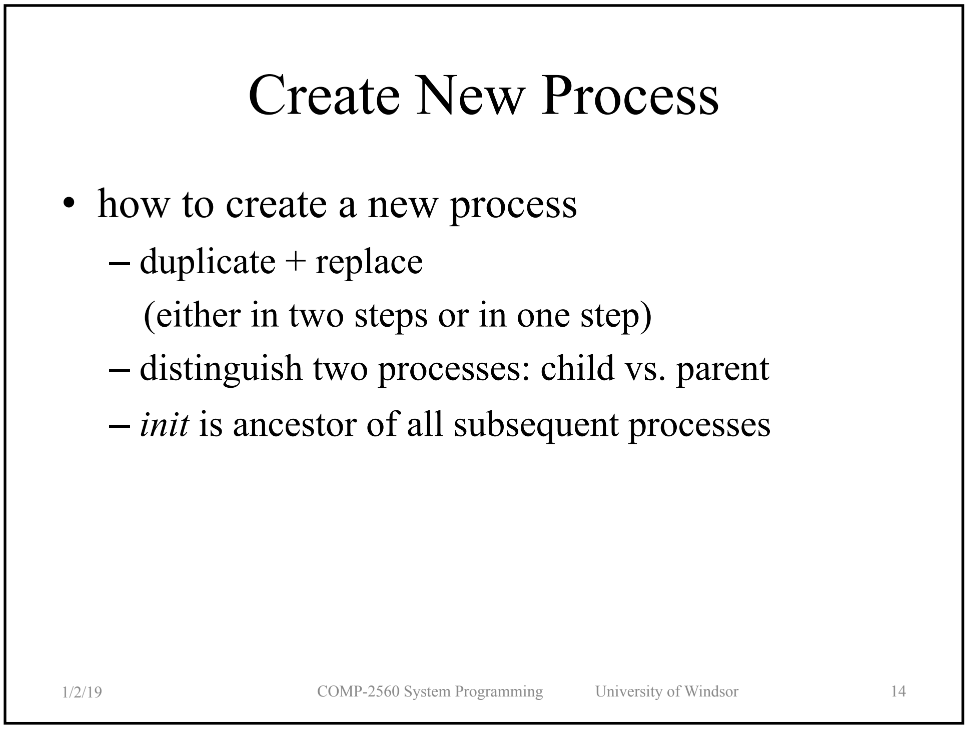
orphan and zombie process

differentiating process: exec()

changing directory: chdir()

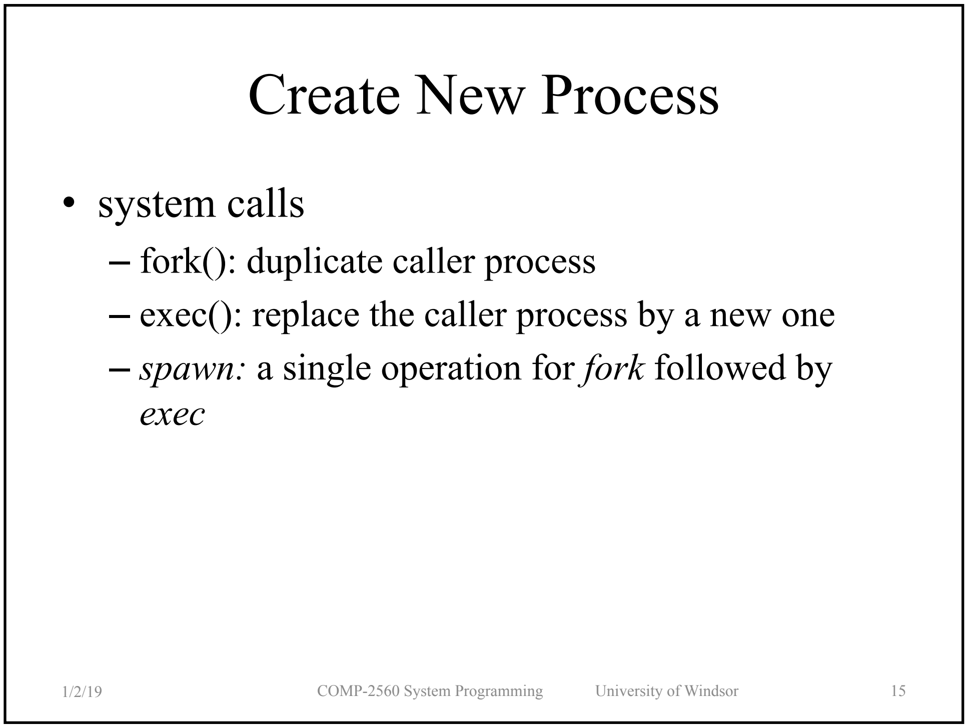
system scheduling priority

process group



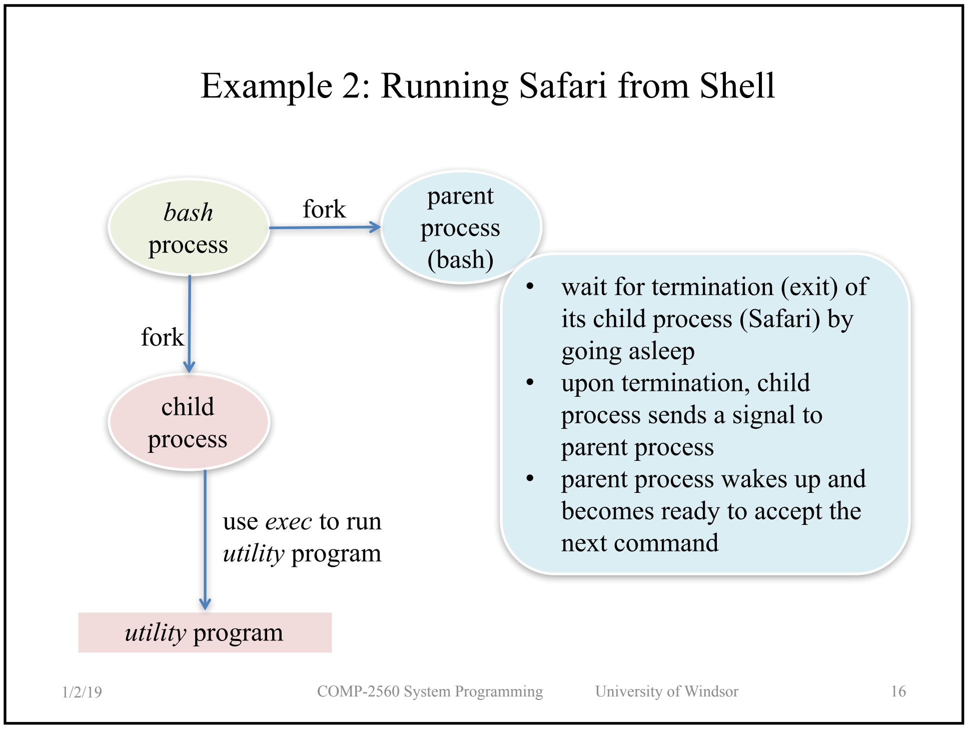
Create New Process

- how to create a new process
  - duplicate + replace
    (either in two steps or in one step)
  - distinguish two processes: child vs. parent
  - *init* is ancestor of all subsequent processes



Create New Process

- system calls
  - fork(): duplicate caller process
  - exec(): replace the caller process by a new one
  - *spawn:* a single operation for *fork* followed by *exec*



Example 2: Running Safari from Shell

- wait for termination (exit) of its child process (Safari) by going asleep
- upon termination, child process sends a signal to parent process
- parent process wakes up and becomes ready to accept the next command

System Call fork()

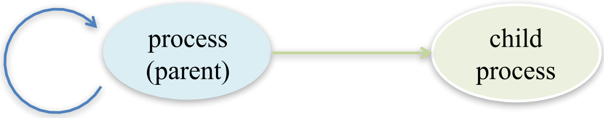
- synopsis

  pid_t fork(void)

- what does it do?
  - try to duplicate caller process
- what does it return?
  - if successful ?
  - if not successful ?

System Call fork()

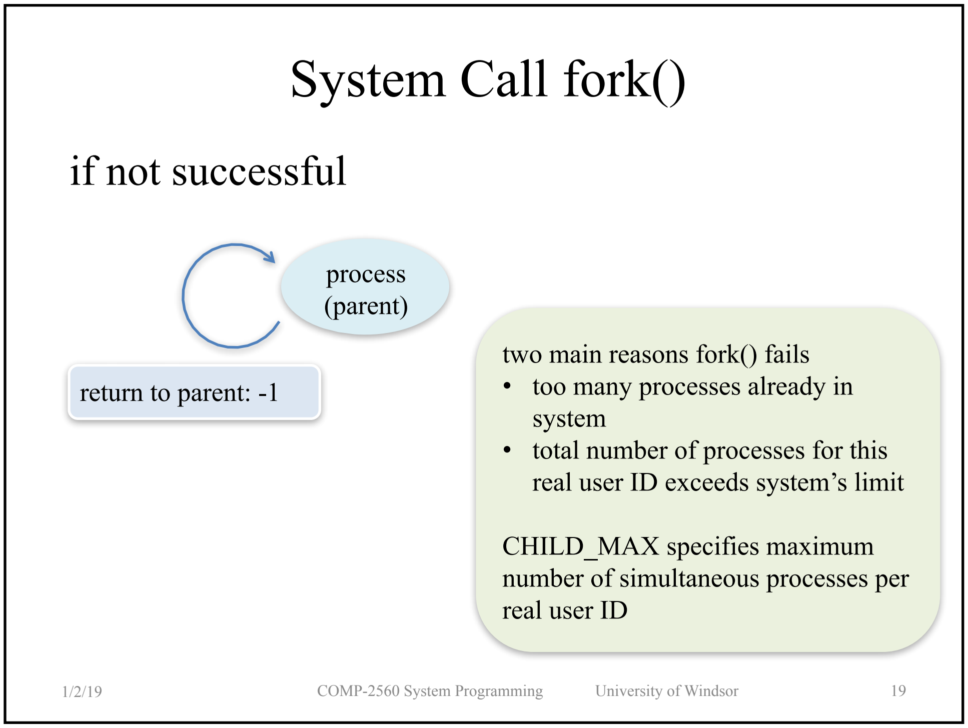
if successful



return to parent: PID of child process

so parent knows the child

return to child process: 0
- PID 0 is preserved for use by kernel
- child can always get to know parent using *getppid*



System Call fork()

if not successful

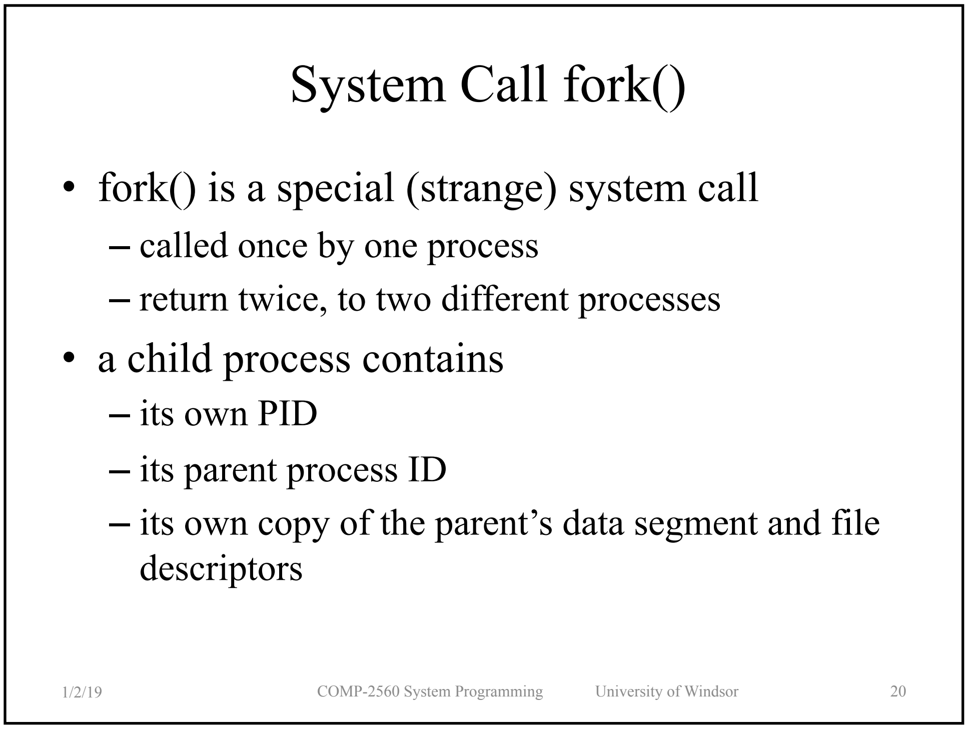
process
(parent)

return to parent: -1

two main reasons fork() fails

- too many processes already in system
- total number of processes for this real user ID exceeds system’s limit

CHILD_MAX specifies maximum number of simultaneous processes per real user ID



System Call fork()

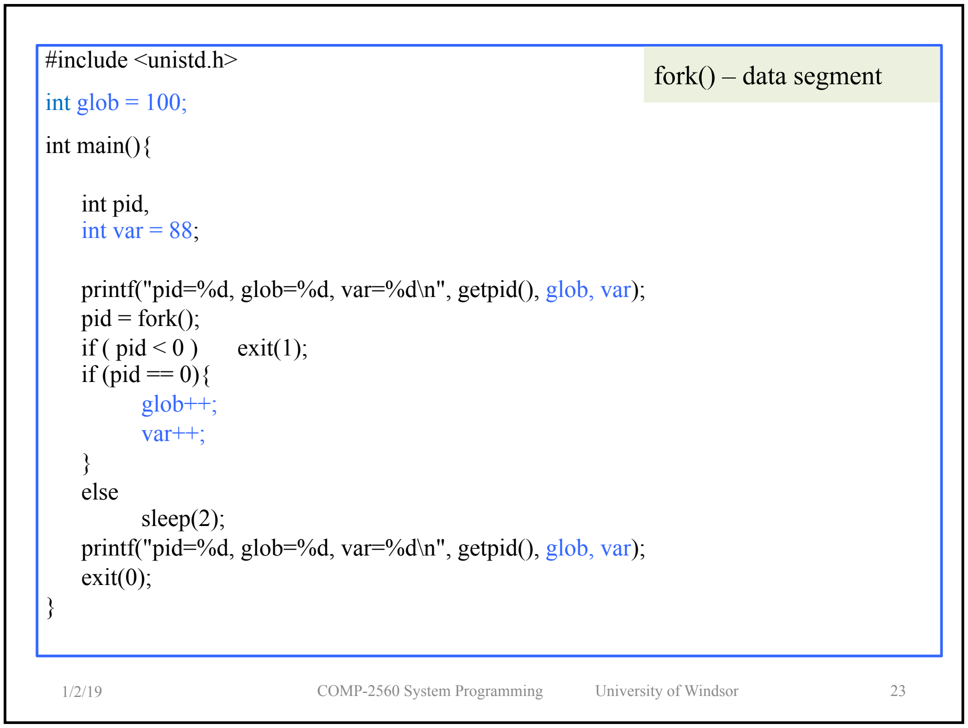
- fork() is a special (strange) system call
  - called once by one process
  - return twice, to two different processes
- a child process contains
  - its own PID
  - its parent process ID
  - its own copy of the parent’s data segment and file descriptors

System Call fork()

- both parent and child resume execution
  - who starts execution first ?

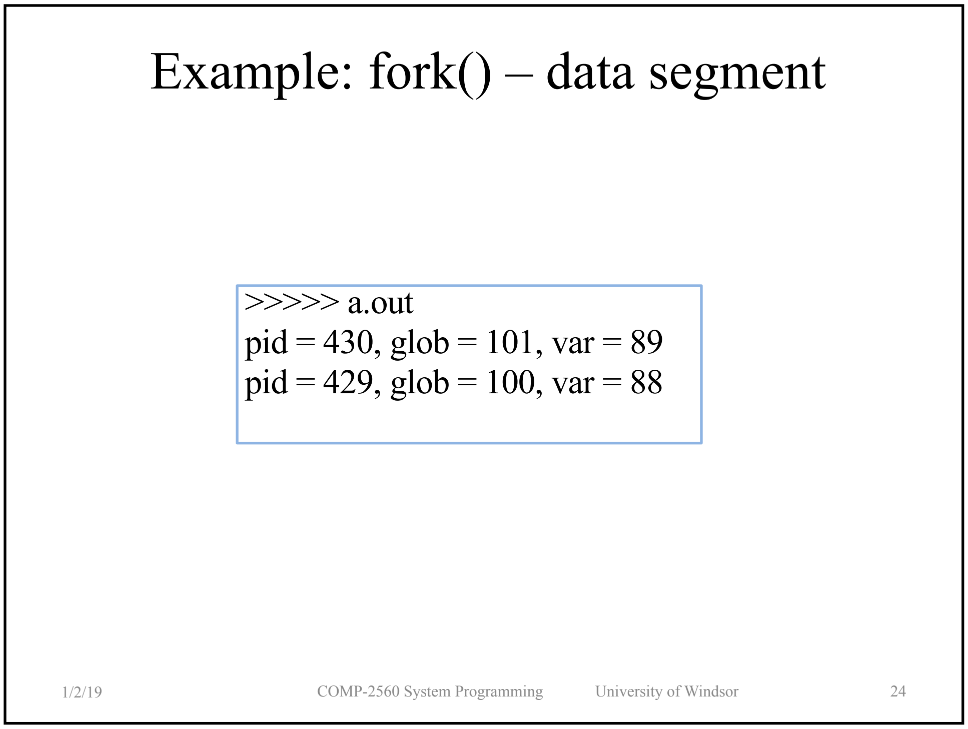
who is executing?

```
#include <unistd.h>
int main(int argc, char *argv[]){
    int npid;
    printf("Initially, PID = %d\n", getpid());
    npid = fork();
    if(npid == -1) {
          perror("impossible to fork");
          exit(1);
    }
    printf("my npid =%d, my PID =%d\n", npid, getpid())
    exit(0);
}
```



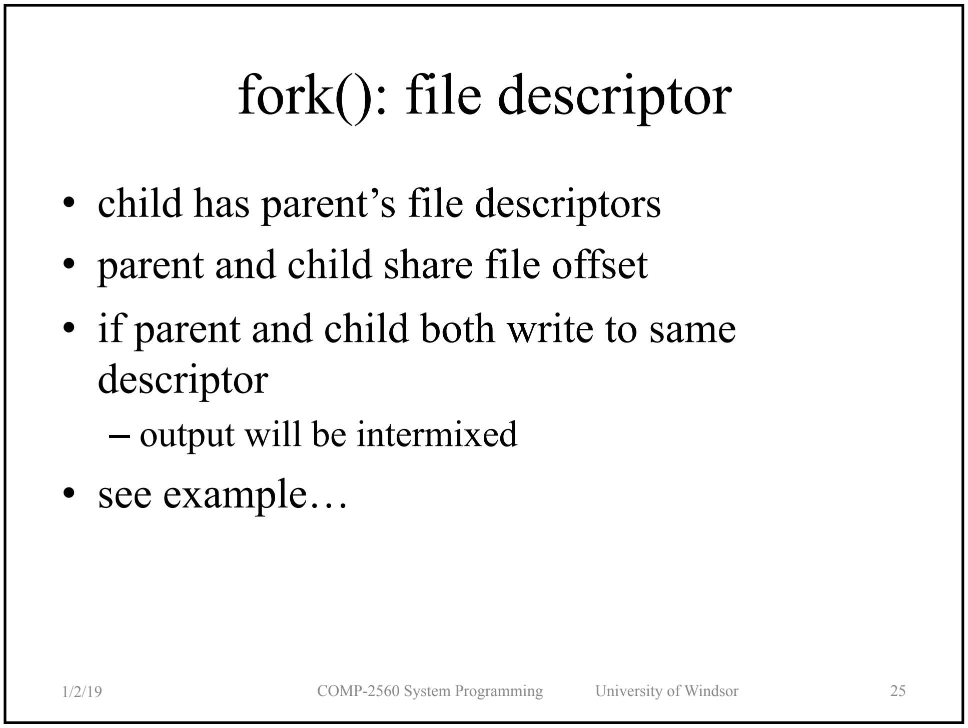
fork() – data segment

```
#include <unistd.h>
int glob = 100;
int main(){

     int pid,
     int var = 88;

     printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);
     pid = fork();
     if ( pid < 0 )      exit(1);
     if (pid == 0){
             glob++;
             var++;
     }
     else
             sleep(2);
     printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);
     exit(0);
}
```



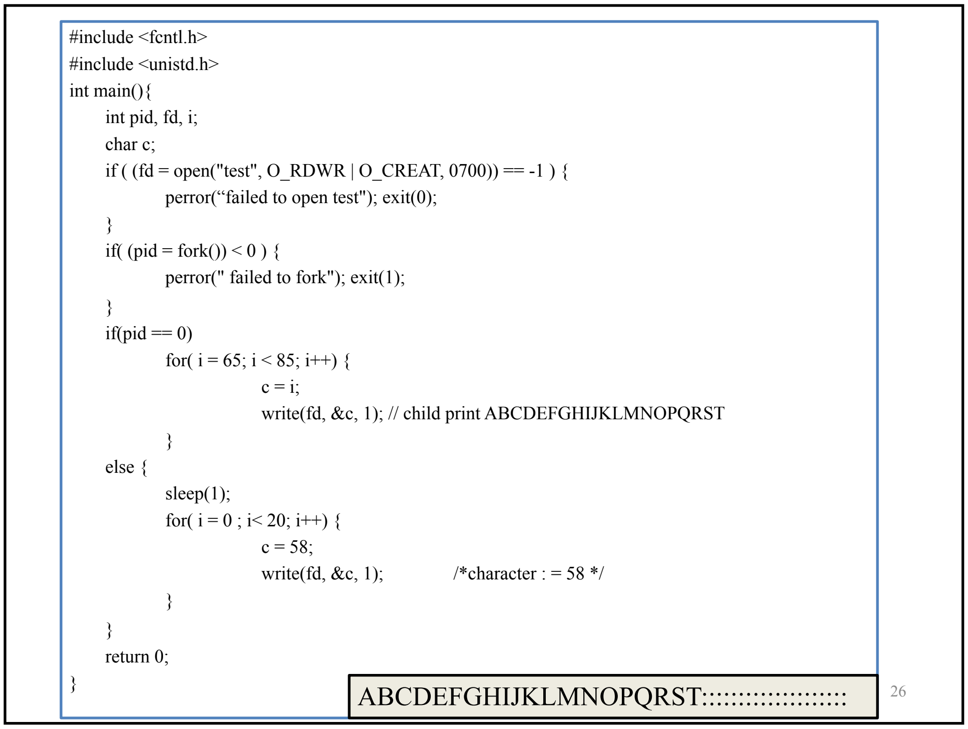
Example: fork() – data segment

```
>>>>> a.out
pid = 430, glob = 101, var = 89
pid = 429, glob = 100, var = 88
```



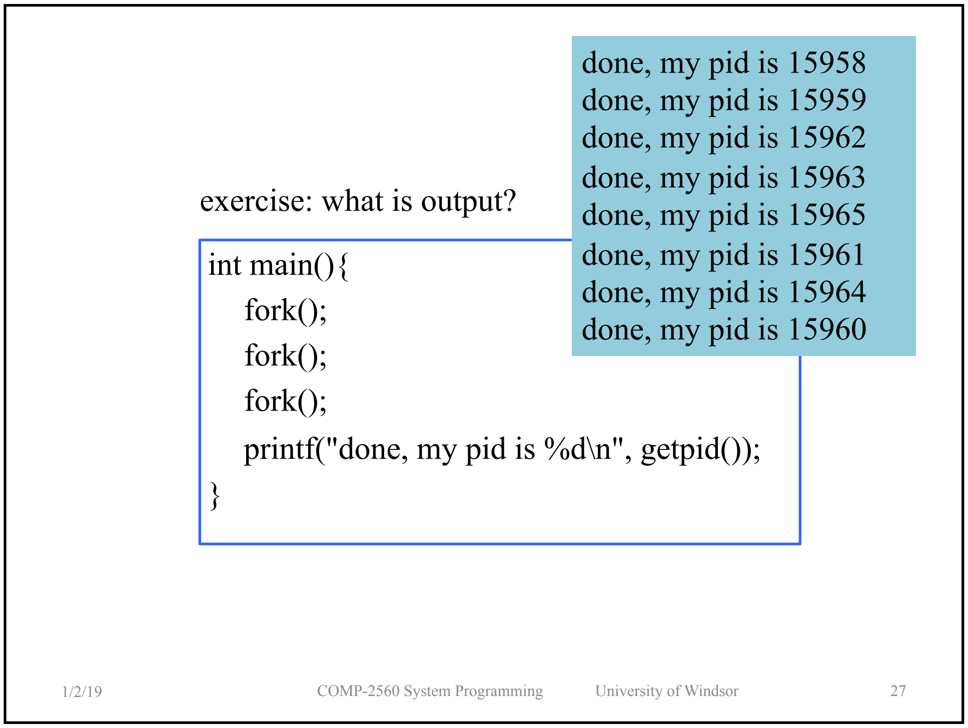
fork(): file descriptor

- child has parent’s file descriptors
- parent and child share file offset
- if parent and child both write to same descriptor
  - output will be intermixed
- see example…



```
#include <fcntl.h>
#include <unistd.h>
int main(){
        int pid, fd, i;
        char c;
        if ( (fd = open("test", O_RDWR | O_CREAT, 0700)) == -1 ) {
                perror(“failed to open test"); exit(0);
        }
        if( (pid = fork()) < 0 ) {
                perror(" failed to fork"); exit(1);
        }
        if(pid == 0)
                for( i = 65; i < 85; i++) {
                        c = i;
                        write(fd, &c, 1); // child print ABCDEFGHIJKLMNOPQRST
                }
        else {
                sleep(1);
                for( i = 0 ; i< 20; i++) {
                        c = 58;
                        write(fd, &c, 1);          /*character : = 58 */
                }
        }
        return 0;
}
```

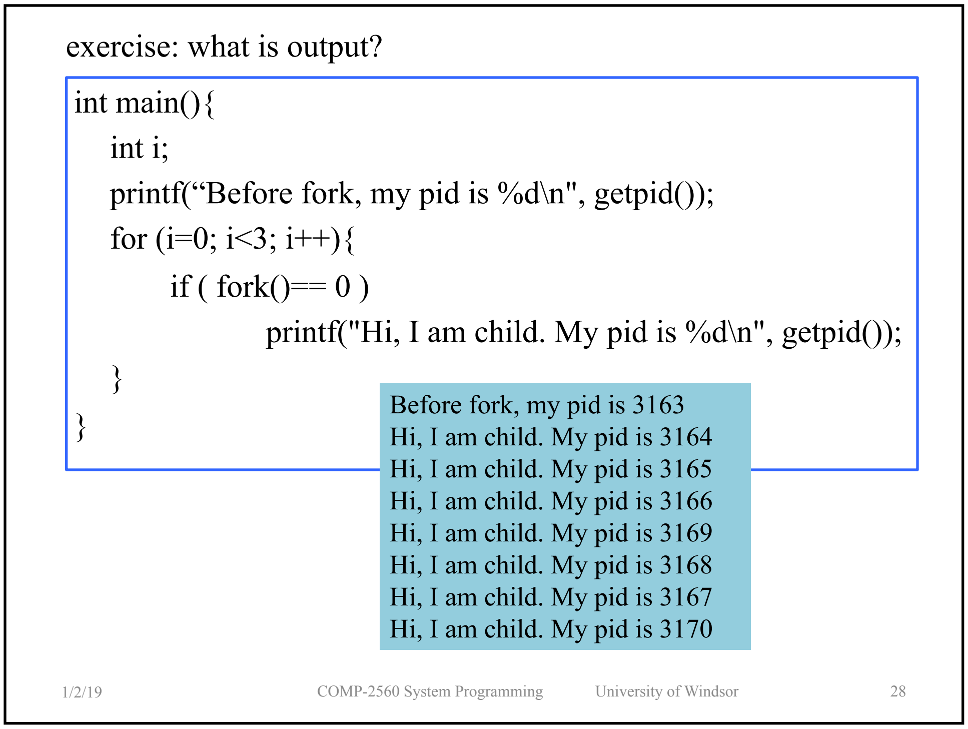
```
ABCDEFGHIJKLMNOPQRST::::::::::::::::::::
```



exercise: what is output?

```
int main(){
    fork();
    fork();
    fork();
    printf("done, my pid is %d\n", getpid());
}
```

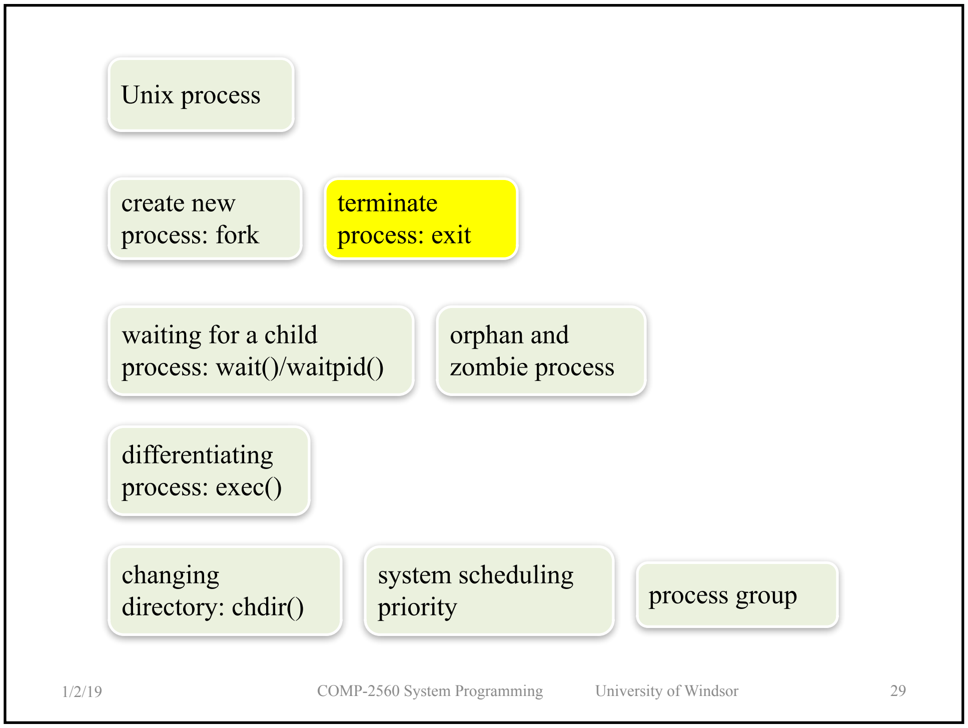
```
done, my pid is 15958
done, my pid is 15959
done, my pid is 15962
done, my pid is 15963
done, my pid is 15965
done, my pid is 15961
done, my pid is 15964
done, my pid is 15960
```



exercise: what is output?

```
int main(){
    int i;
    printf("Before fork, my pid is %d\n", getpid());
    for (i=0; i<3; i++){
        if ( fork()== 0 )
                printf("Hi, I am child. My pid is %d\n", getpid());
    }
}
```

```
Before fork, my pid is 3163
Hi, I am child. My pid is 3164
Hi, I am child. My pid is 3165
Hi, I am child. My pid is 3166
Hi, I am child. My pid is 3169
Hi, I am child. My pid is 3168
Hi, I am child. My pid is 3167
Hi, I am child. My pid is 3170
```



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

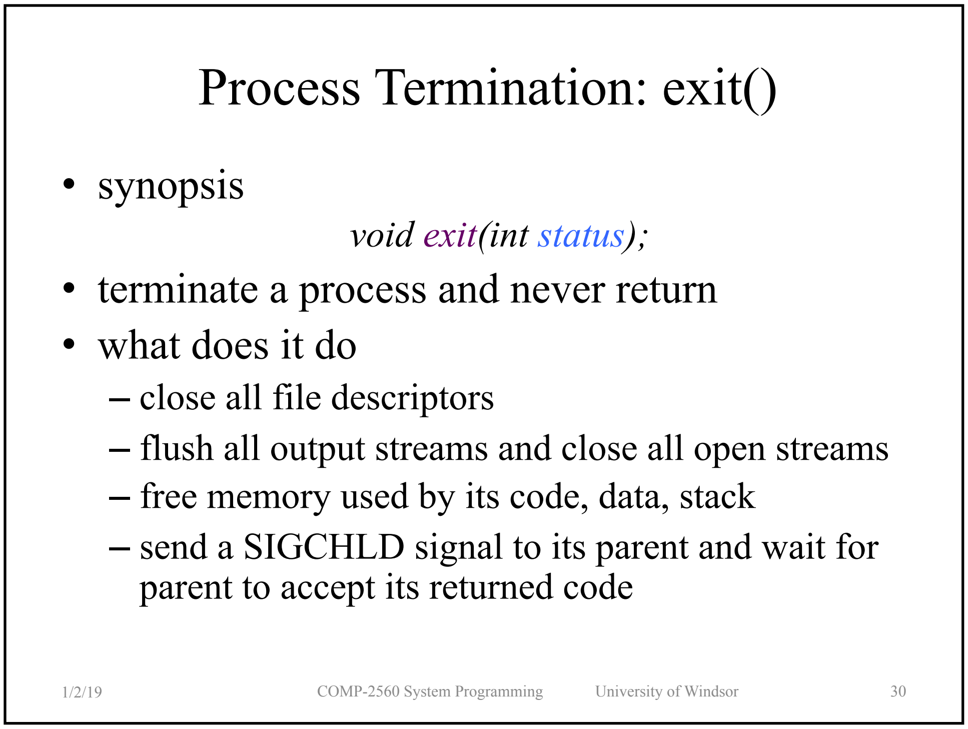
orphan and zombie process

differentiating process: exec()

changing directory: chdir()

system scheduling priority

process group

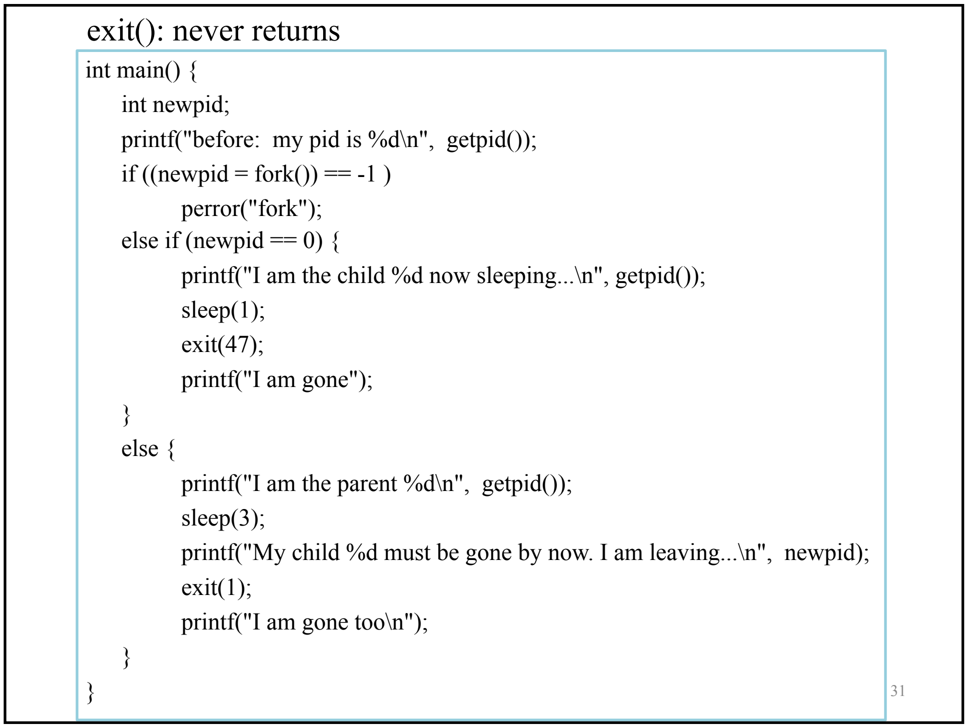


Process Termination: exit()

- synopsis

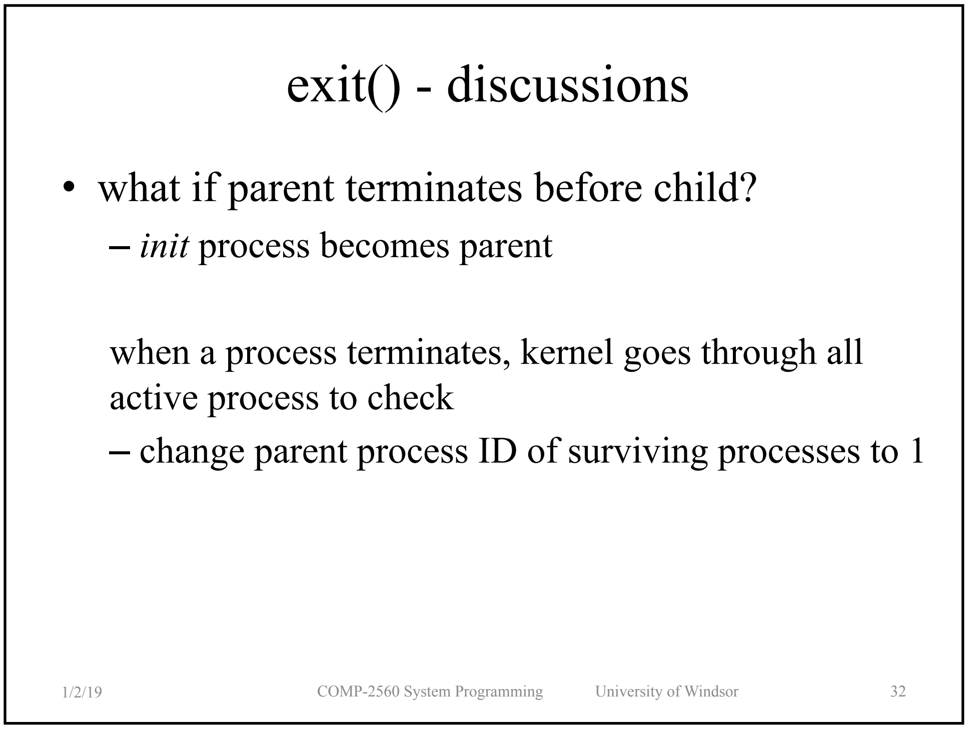
  void exit(int status);

- terminate a process and never return
- what does it do
  - close all file descriptors
  - flush all output streams and close all open streams
  - free memory used by its code, data, stack
  - send a SIGCHLD signal to its parent and wait for parent to accept its returned code



exit(): never returns

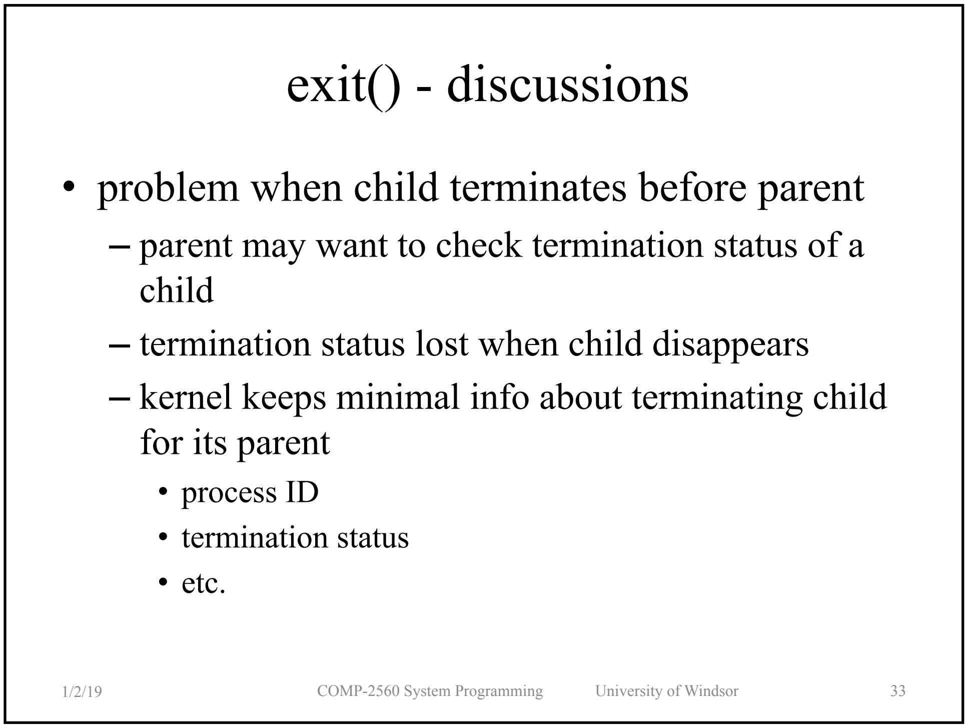
```
int main() {
     int newpid;
     printf("before:  my pid is %d\n",  getpid());
     if ((newpid = fork()) == -1 )
              perror("fork");
     else if (newpid == 0) {
              printf("I am the child %d now sleeping...\n", getpid());
              sleep(1);
              exit(47);
              printf("I am gone");
     }
     else {
              printf("I am the parent %d\n",  getpid());
              sleep(3);
              printf("My child %d must be gone by now. I am leaving...\n",  newpid);
              exit(1);
              printf("I am gone too\n");
     }
}
```



exit() - discussions

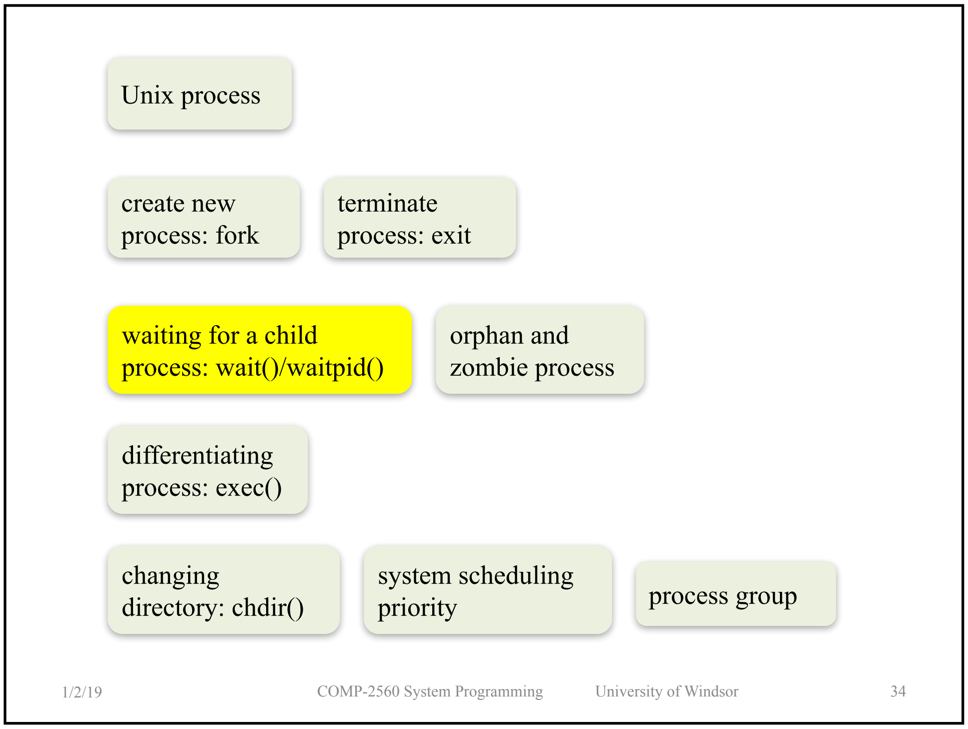
- what if parent terminates before child?
  - *init* process becomes parent

  when a process terminates, kernel goes through all active process to check
  - change parent process ID of surviving processes to 1



exit() - discussions

- problem when child terminates before parent
  - parent may want to check termination status of a child
  - termination status lost when child disappears
  - kernel keeps minimal info about terminating child for its parent
    - process ID
    - termination status
    - etc.



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

orphan and zombie process

differentiating process: exec()

changing directory: chdir()

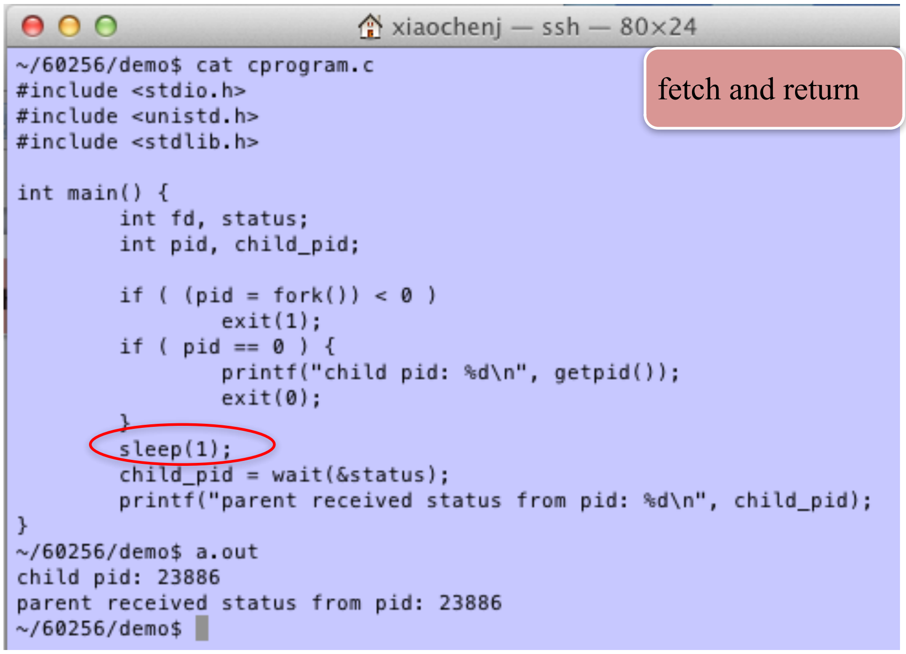
system scheduling priority

process group

wait()

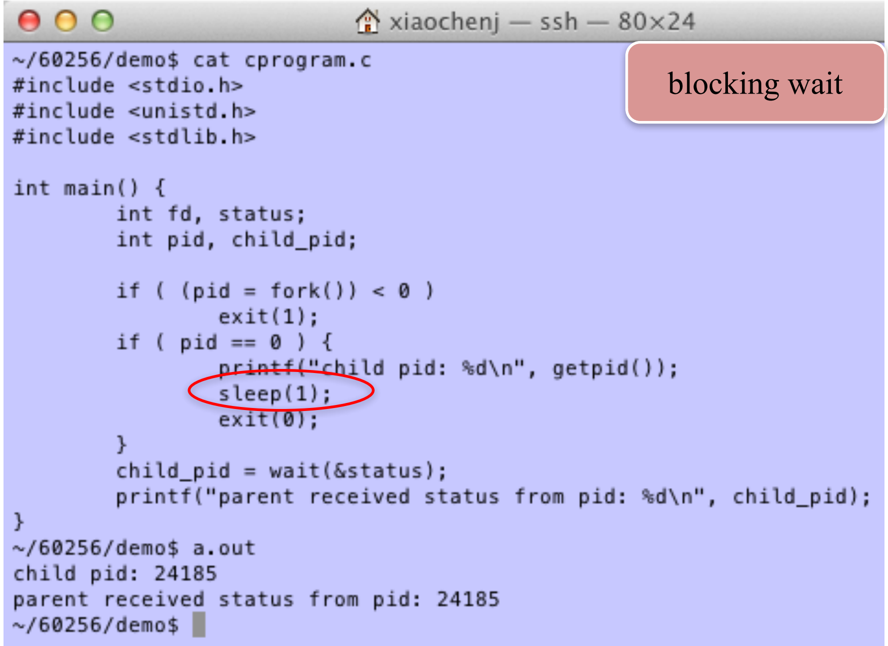
```
# include <sys/wait.h>
pid_t wait(int *statloc);
                              return: process ID if ok, 0 or -1 on error
```

- what does *wait* do
  - if all children are running, get blocked
  - if no child process, return with error
  - if a child terminated and waiting for its termination status to be fetched, return with status



fetch and return

```
~/60256/demo$ cat cprogram.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
        int fd, status;
        int pid, child_pid;

        if ( (pid = fork()) < 0 )
                exit(1);
        if ( pid == 0 ) {
                printf("child pid: %d\n", getpid());
                exit(0);
        }
        sleep(1);
        child_pid = wait(&status);
        printf("parent received status from pid: %d\n", child_pid);
}
~/60256/demo$ a.out
child pid: 23886
parent received status from pid: 23886
~/60256/demo$ 
```



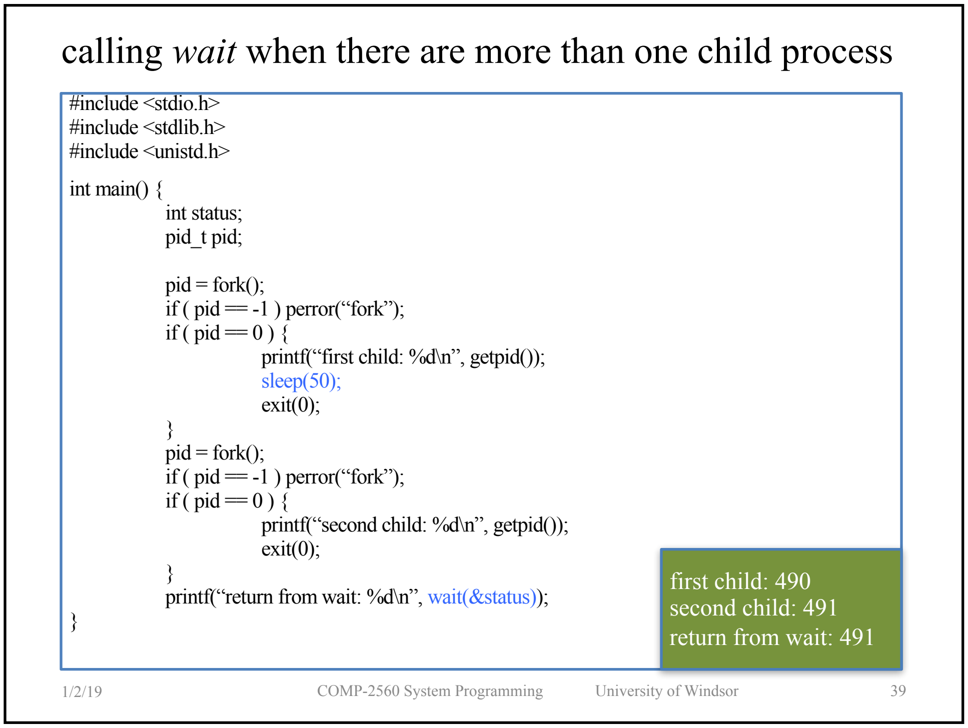
blocking wait

```
~/60256/demo$ cat cprogram.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
        int fd, status;
        int pid, child_pid;

        if ( (pid = fork()) < 0 )
                exit(1);
        if ( pid == 0 ) {
                printf("child pid: %d\n", getpid());
                sleep(1);
                exit(0);
        }
        child_pid = wait(&status);
        printf("parent received status from pid: %d\n", child_pid);
}
~/60256/demo$ a.out
child pid: 24185
parent received status from pid: 24185
~/60256/demo$ 
```

wait()

```
# include <sys/wait.h>
pid_t wait(int *statloc);
                    return: process ID if ok, 0 or -1 on error
```

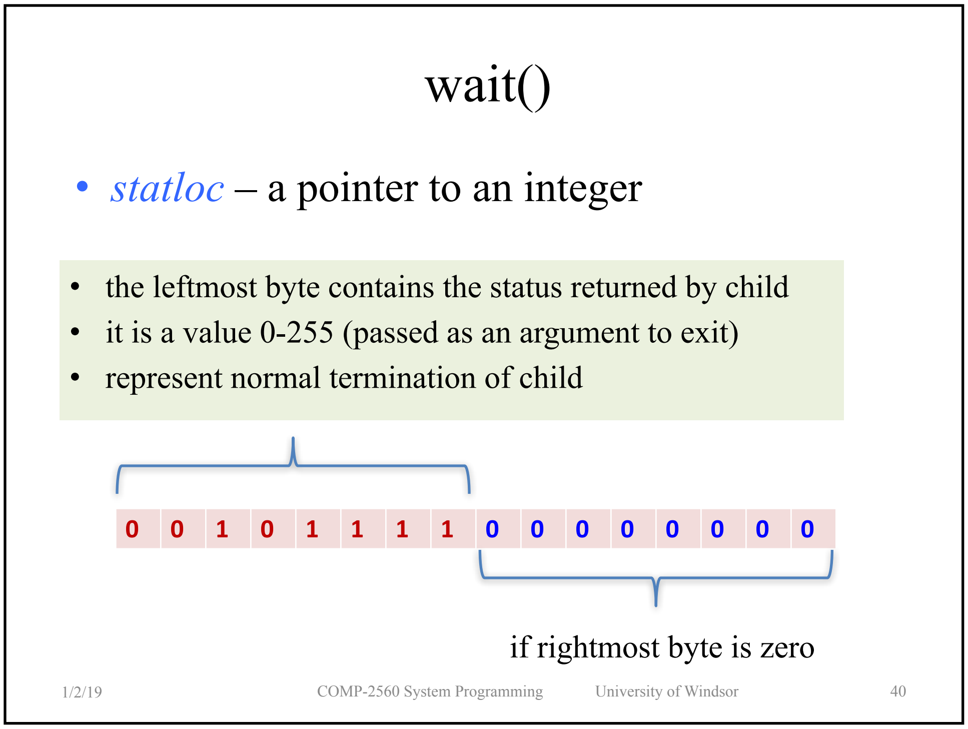
- with return PID value of *wait*, we can tell which child terminated



calling *wait* when there are more than one child process

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

int main() {
        int status;
        pid_t pid;

        pid = fork();
        if ( pid == -1 ) perror(“fork”);
        if ( pid == 0 ) {
                printf(“first child: %d\n”, getpid());
                sleep(50);
                exit(0);
        }
        pid = fork();
        if ( pid == -1 ) perror(“fork”);
        if ( pid == 0 ) {
                printf(“second child: %d\n”, getpid());
                exit(0);
        }
        printf(“return from wait: %d\n”, wait(&status));
}
```

```
first child: 490
second child: 491
return from wait: 491
```



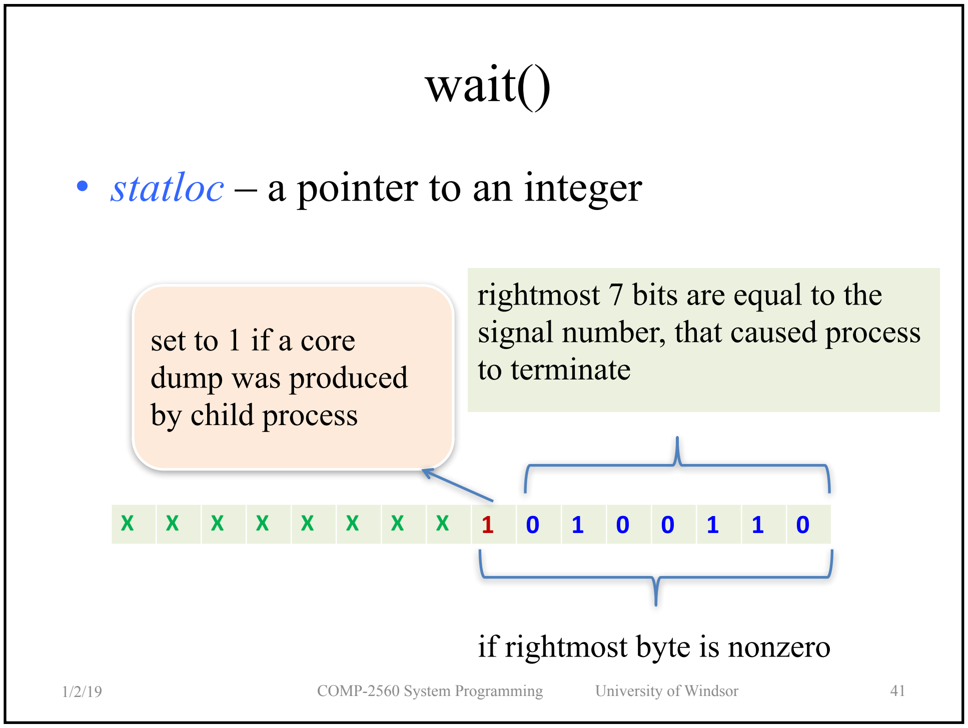
wait()

- *statloc* – a pointer to an integer

- the leftmost byte contains the status returned by child
- it is a value 0-255 (passed as an argument to exit)
- represent normal termination of child

0	0	1	0	1	1	1	1	0	0	0	0	0	0	0	0

if rightmost byte is zero



wait()

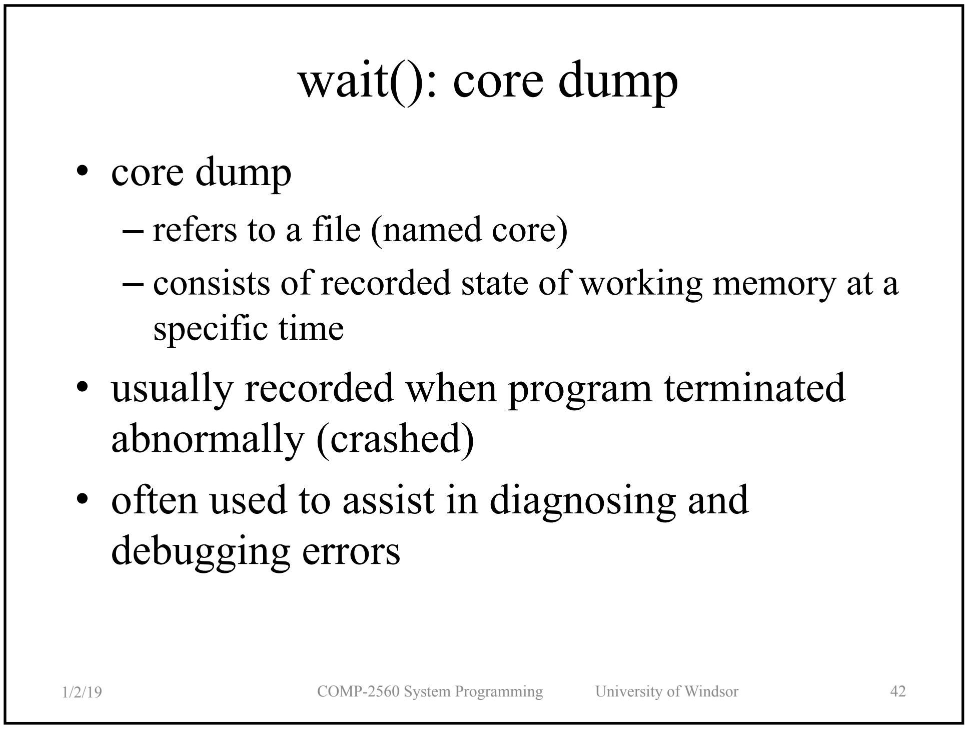
- *statloc* – a pointer to an integer

set to 1 if a core dump was produced by child process

rightmost 7 bits are equal to the signal number, that caused process to terminate

x	x	x	x	x	x	x	x	1	0	1	0	0	1	1	0

if rightmost byte is nonzero



wait(): core dump

- core dump
  - refers to a file (named core)
  - consists of recorded state of working memory at a specific time
- usually recorded when program terminated abnormally (crashed)
- often used to assist in diagnosing and debugging errors

Example

- let child process exit with 47
- the parent will have status value: 2f00 (12032)

abort()

- synopsis

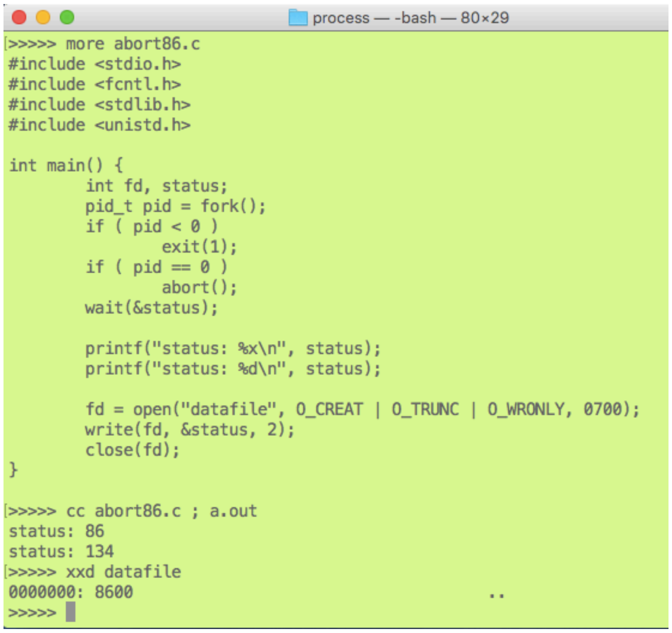
  void abort(void)

- declared in <stdlib.h>
- causes abnormal process termination to occur
- it sends signal SIGABRT (6) to the parent process
- causes a core dump

Example

- above example again
- let child process call abort()
- the parent will have status value: 0086 (SIGABRT 6, with core dump)
- to enable core dump on your machine, you can set

> ulimit –c unlimited



```
[>>>>> more abort86.c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
        int fd, status;
        pid_t pid = fork();
        if ( pid < 0 )
                exit(1);
        if ( pid == 0 )
                abort();
        wait(&status);

        printf("status: %x\n", status);
        printf("status: %d\n", status);

        fd = open("datafile", O_CREAT | O_TRUNC | O_WRONLY, 0700);
        write(fd, &status, 2);
        close(fd);
}

[>>>>> cc abort86.c ; a.out
status: 86
status: 134
[>>>>> xxd datafile
0000000: 8600                                     ..
>>>>> 
```

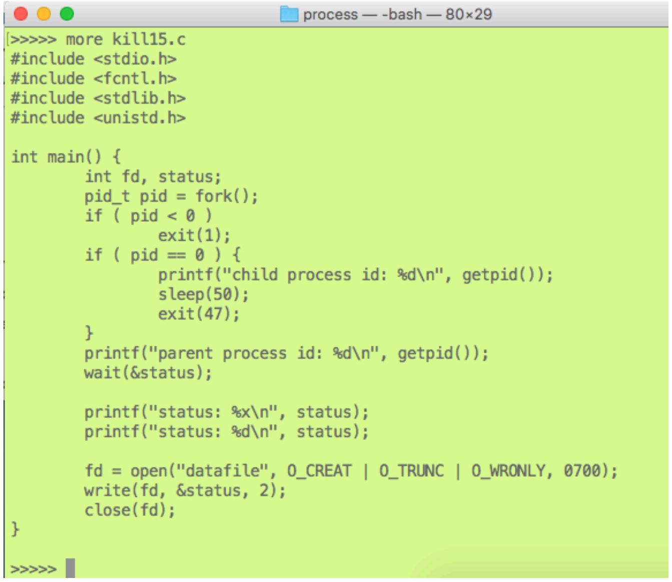
Example

- now, before the child makes exit call, let us terminate the child process

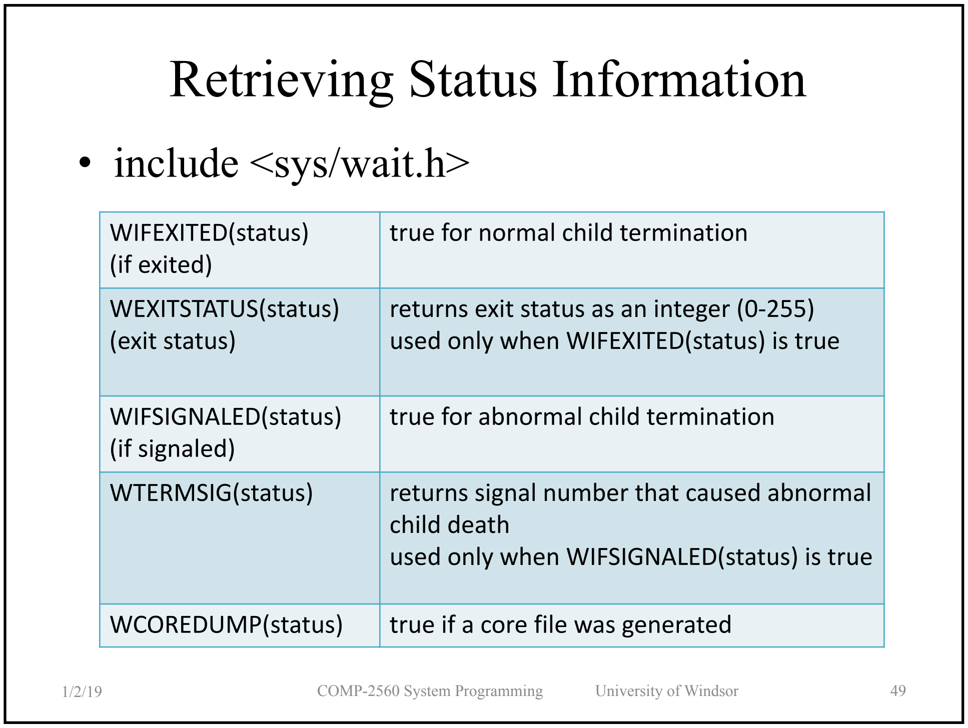
  > kill -15 child-id

- the parent will have status value: 000f (SIGTERM 15)

child process: sleep and get external signal to terminate



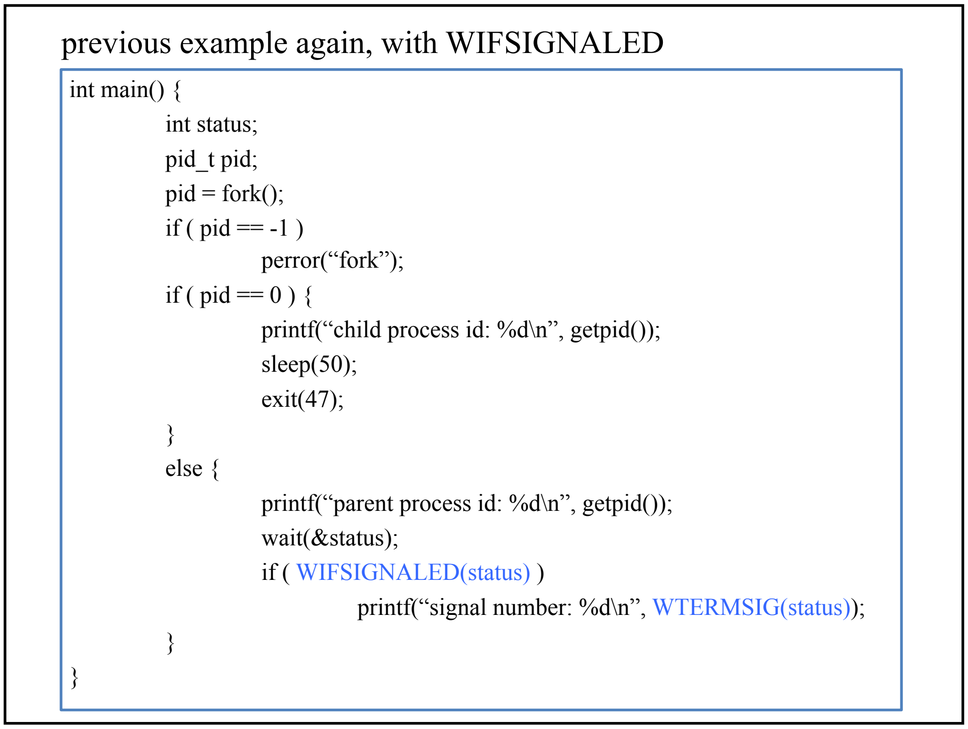
```
>>>>> more kill15.c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
        int fd, status;
        pid_t pid = fork();
        if ( pid < 0 )
                exit(1);
        if ( pid == 0 ) {
                printf("child process id: %d\n", getpid());
                sleep(50);
                exit(47);
        }
        printf("parent process id: %d\n", getpid());
        wait(&status);

        printf("status: %x\n", status);
        printf("status: %d\n", status);

        fd = open("datafile", O_CREAT | O_TRUNC | O_WRONLY, 0700);
        write(fd, &status, 2);
        close(fd);
}

>>>>> 
```



Retrieving Status Information

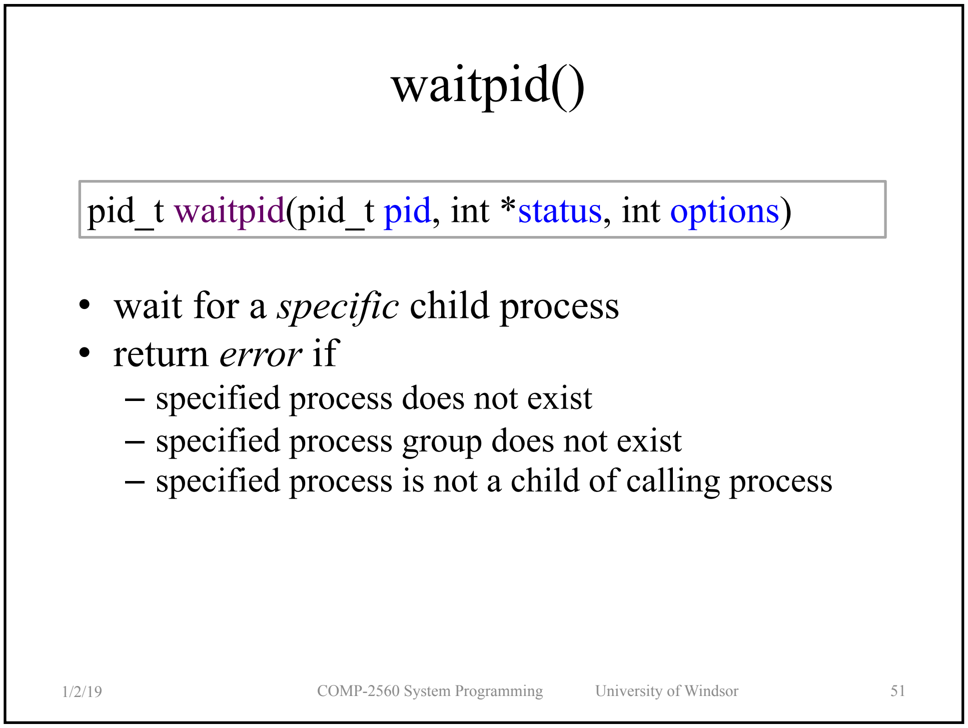
- include <sys/wait.h>

WIFEXITED(status) (if exited)	true for normal child termination
WEXITSTATUS(status) (exit status)	returns exit status as an integer (0-255) used only when WIFEXITED(status) is true
WIFSIGNALED(status) (if signaled)	true for abnormal child termination
WTERMSIG(status)	returns signal number that caused abnormal child death used only when WIFSIGNALED(status) is true
WCOREDUMP(status)	true if a core file was generated



previous example again, with WIFSIGNALED

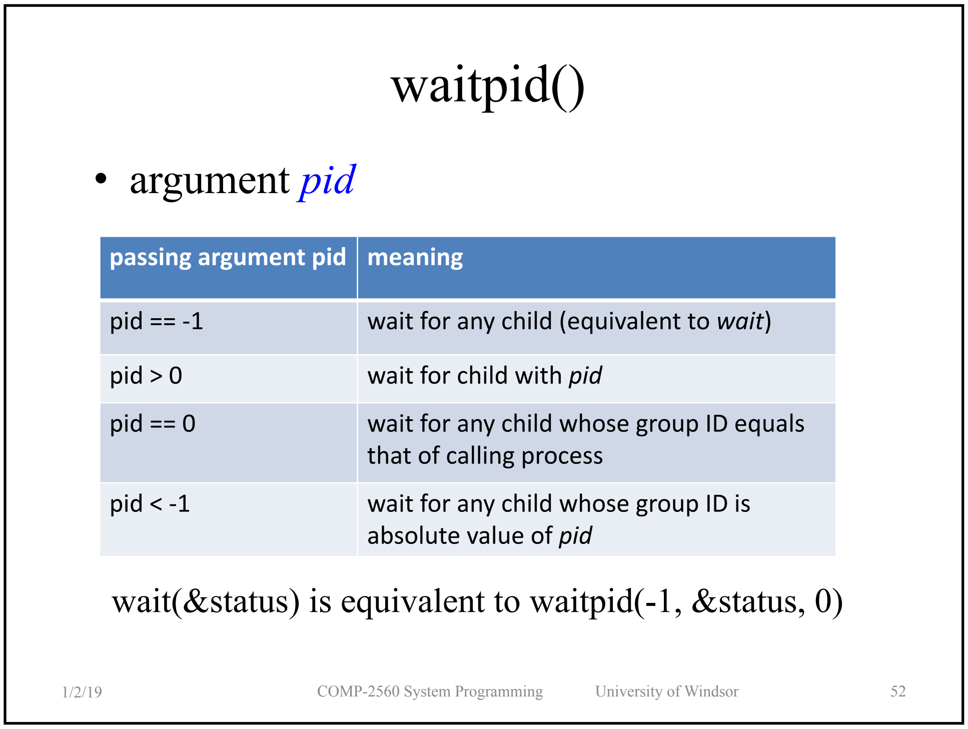
```
int main() {
        int status;
        pid_t pid;
        pid = fork();
        if ( pid == -1 )
                perror(“fork”);
        if ( pid == 0 ) {
                printf(“child process id: %d\n”, getpid());
                sleep(50);
                exit(47);
        }
        else {
                printf(“parent process id: %d\n”, getpid());
                wait(&status);
                if ( WIFSIGNALED(status) )
                        printf(“signal number: %d\n”, WTERMSIG(status));
        }
}
```



waitpid()

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

- wait for a *specific* child process
- return *error* if
  - specified process does not exist
  - specified process group does not exist
  - specified process is not a child of calling process

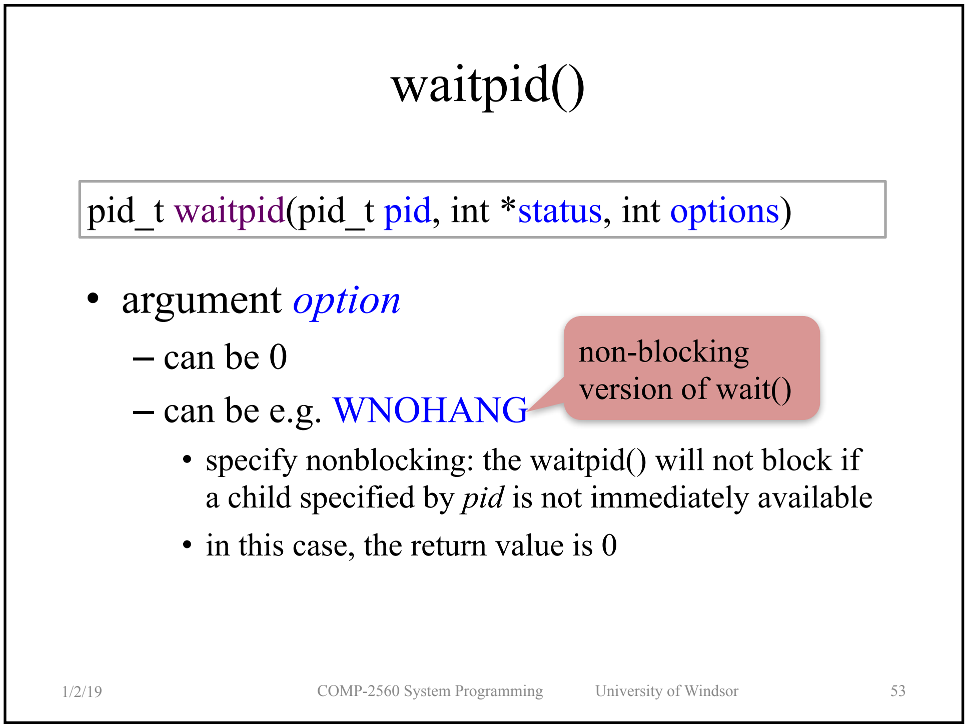


waitpid()

- argument *pid*

passing argument pid	meaning
pid == -1	wait for any child (equivalent to *wait*)
pid > 0	wait for child with *pid*
pid == 0	wait for any child whose group ID equals that of calling process
pid < -1	wait for any child whose group ID is absolute value of *pid*

wait(&status) is equivalent to waitpid(-1, &status, 0)

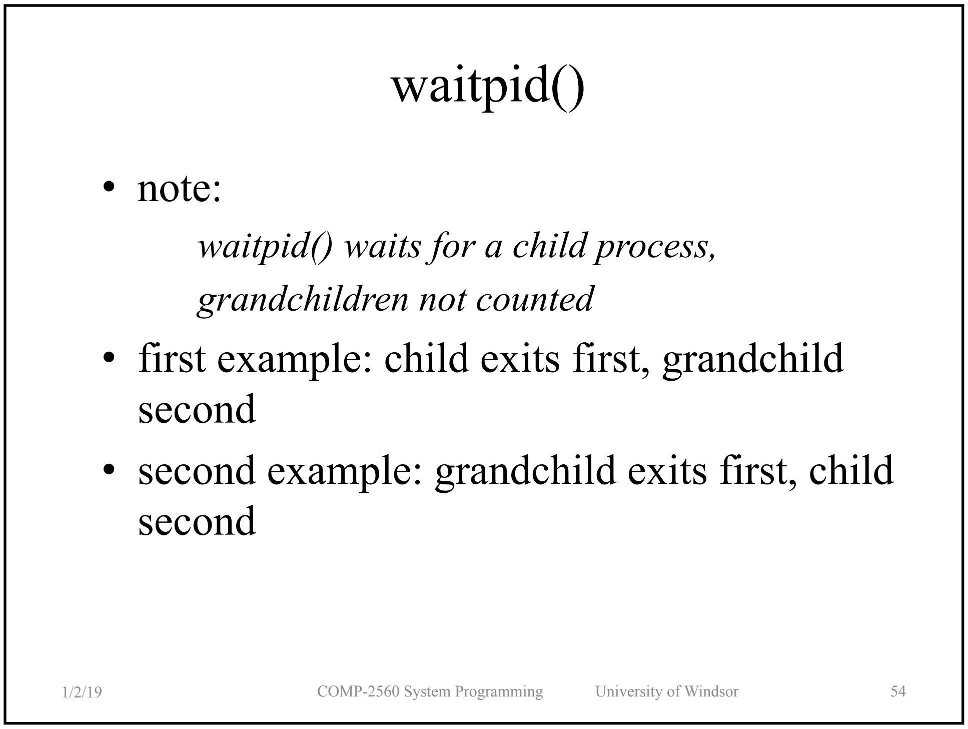


waitpid()

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

- argument *option*
  - can be 0
  - can be e.g. WNOHANG
    - specify nonblocking: the waitpid() will not block if a child specified by *pid* is not immediately available
    - in this case, the return value is 0

non-blocking version of wait()

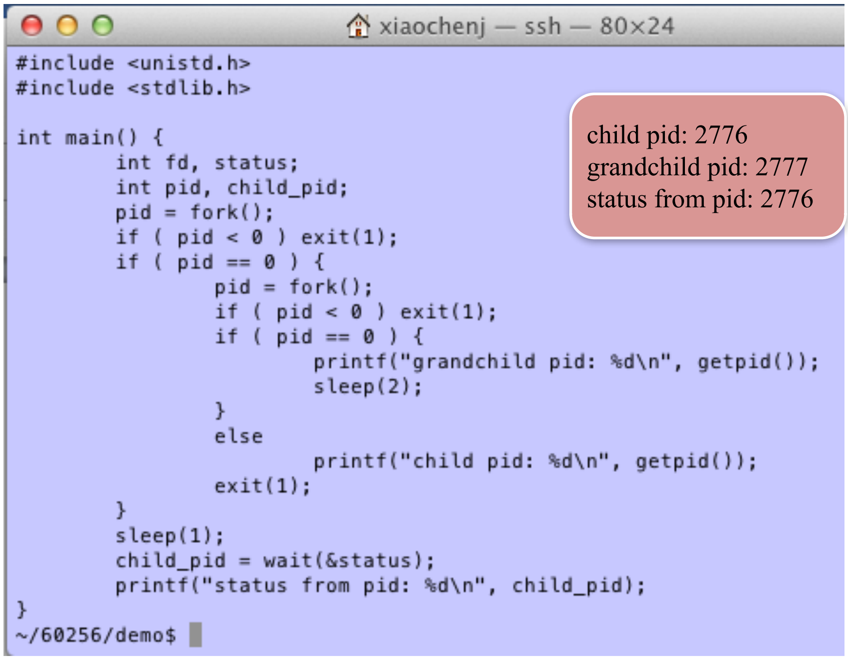


waitpid()

- note:

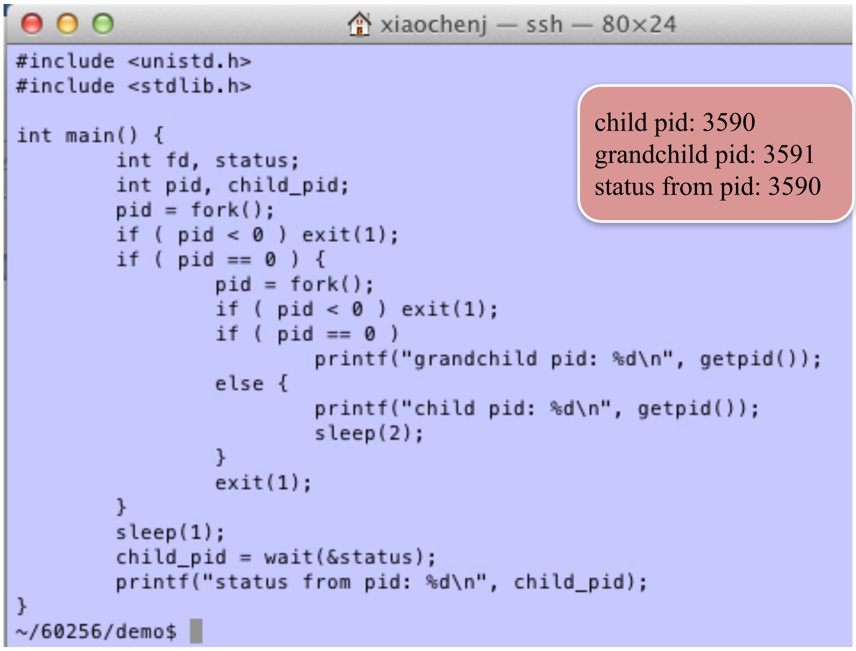
  waitpid() waits for a child process, grandchildren not counted

- first example: child exits first, grandchild second
- second example: grandchild exits first, child second



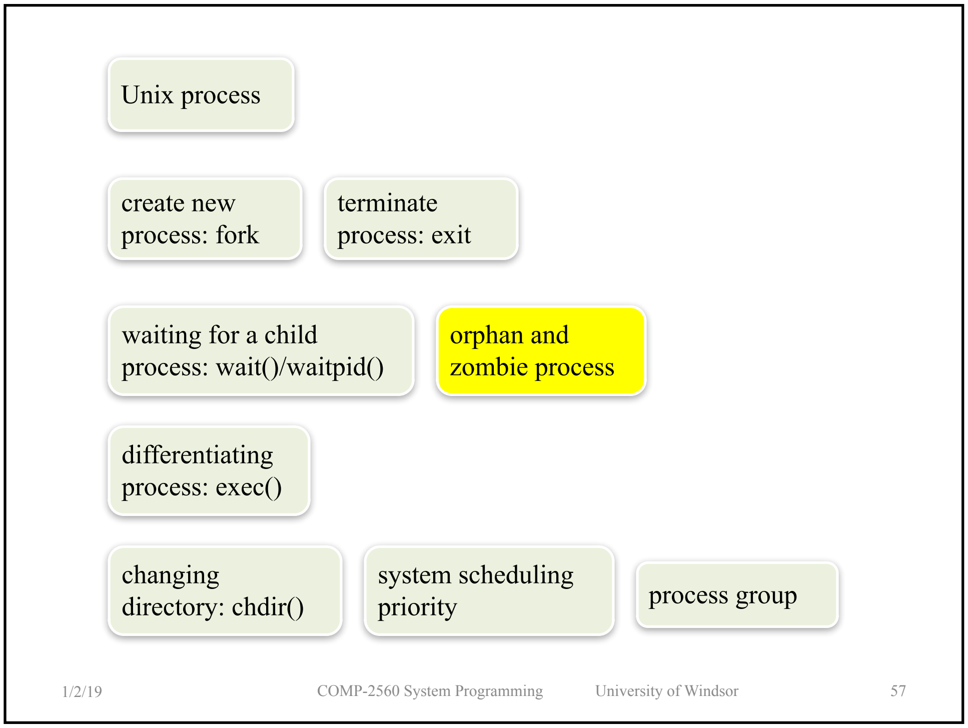
```
#include <unistd.h>
#include <stdlib.h>

int main() {
        int fd, status;
        int pid, child_pid;
        pid = fork();
        if ( pid < 0 ) exit(1);
        if ( pid == 0 ) {
                pid = fork();
                if ( pid < 0 ) exit(1);
                if ( pid == 0 ) {
                        printf("grandchild pid: %d\n", getpid());
                        sleep(2);
                }
                else
                        printf("child pid: %d\n", getpid());
                exit(1);
        }
        sleep(1);
        child_pid = wait(&status);
        printf("status from pid: %d\n", child_pid);
}
~/60256/demo$
```

child pid: 2776
grandchild pid: 2777
status from pid: 2776



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

int main() {
        int fd, status;
        int pid, child_pid;
        pid = fork();
        if ( pid < 0 ) exit(1);
        if ( pid == 0 ) {
                pid = fork();
                if ( pid < 0 ) exit(1);
                if ( pid == 0 )
                        printf("grandchild pid: %d\n", getpid());
                else {
                        printf("child pid: %d\n", getpid());
                        sleep(2);
                }
                exit(1);
        }
        sleep(1);
        child_pid = wait(&status);
        printf("status from pid: %d\n", child_pid);
}
~/60256/demo$
```

child pid: 3590
grandchild pid: 3591
status from pid: 3590



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

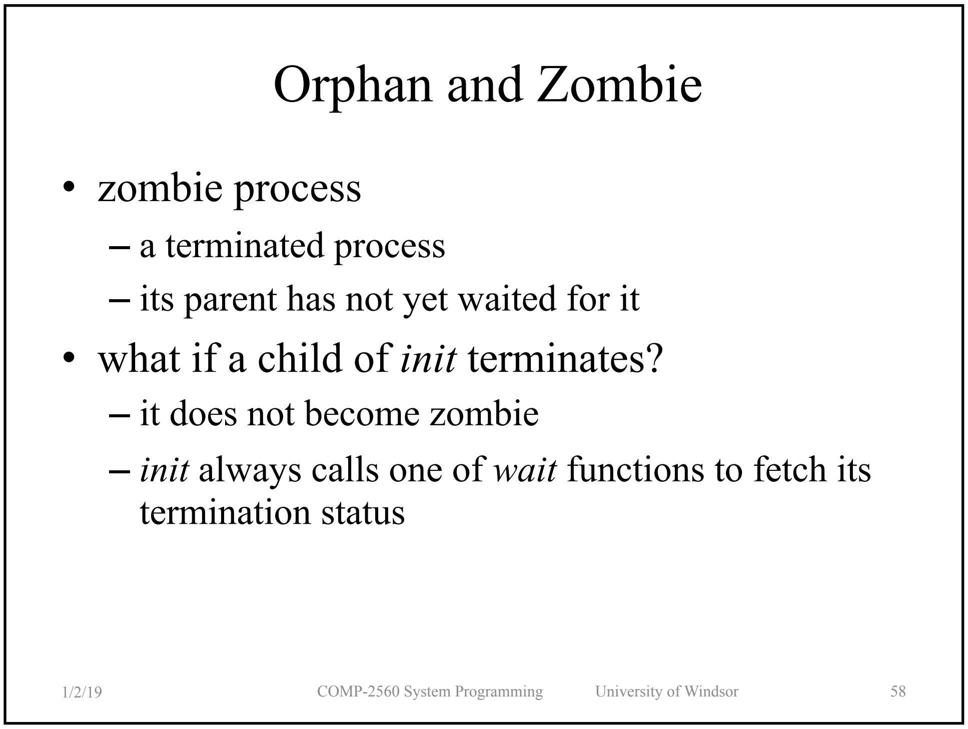
orphan and zombie process

differentiating process: exec()

changing directory: chdir()

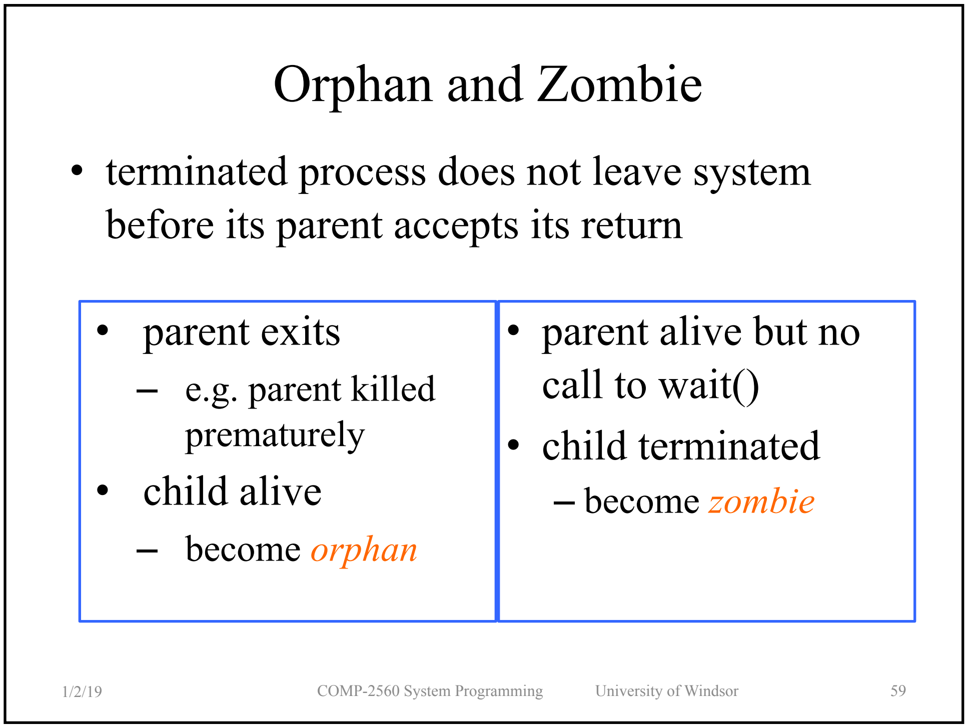
system scheduling priority

process group



Orphan and Zombie

- zombie process
  - a terminated process
  - its parent has not yet waited for it
- what if a child of *init* terminates?
  - it does not become zombie
  - *init* always calls one of *wait* functions to fetch its termination status

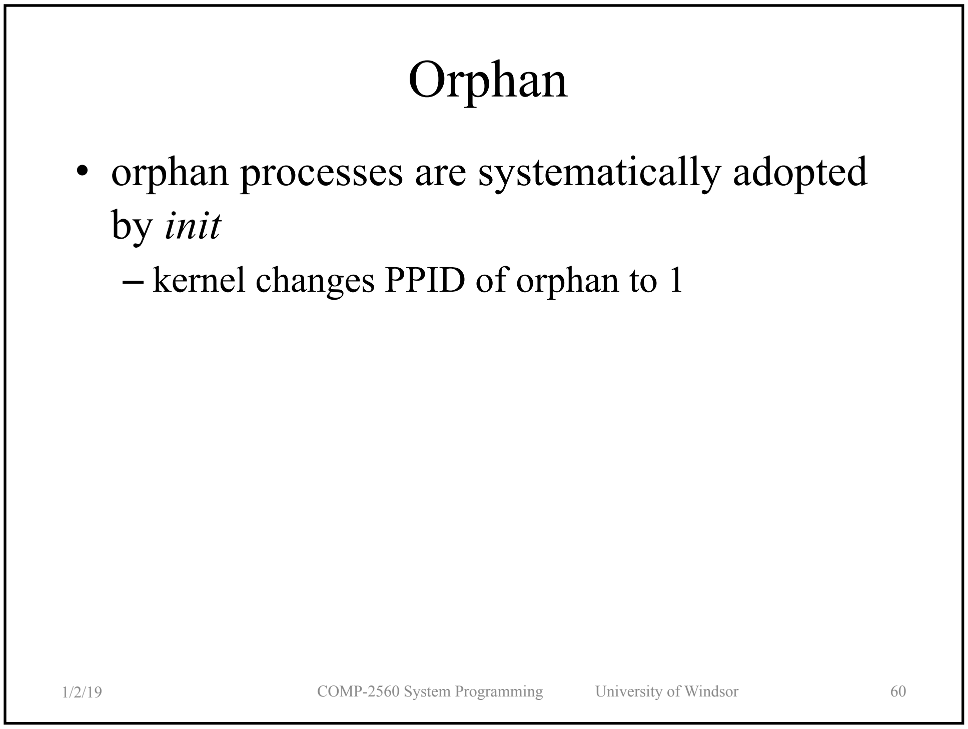


Orphan and Zombie

- terminated process does not leave system before its parent accepts its return

- parent exits
  - e.g. parent killed prematurely
- child alive
  - become *orphan*

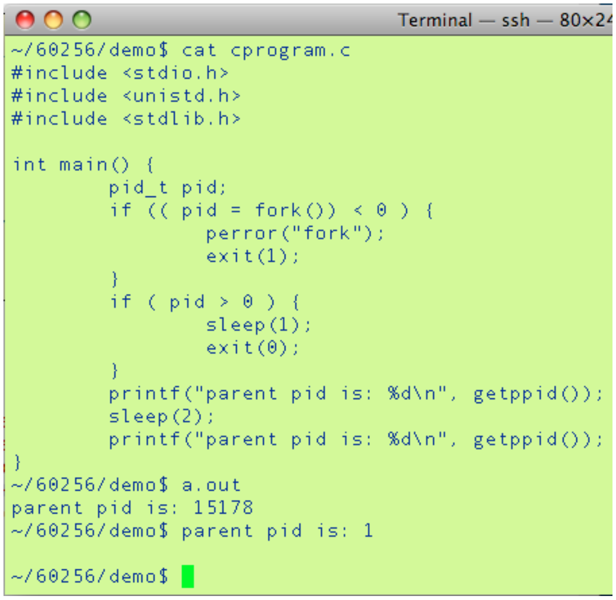
- parent alive but no call to wait()
- child terminated
  - become *zombie*



Orphan

- orphan processes are systematically adopted by *init*
  - kernel changes PPID of orphan to 1

becoming child of *init* process



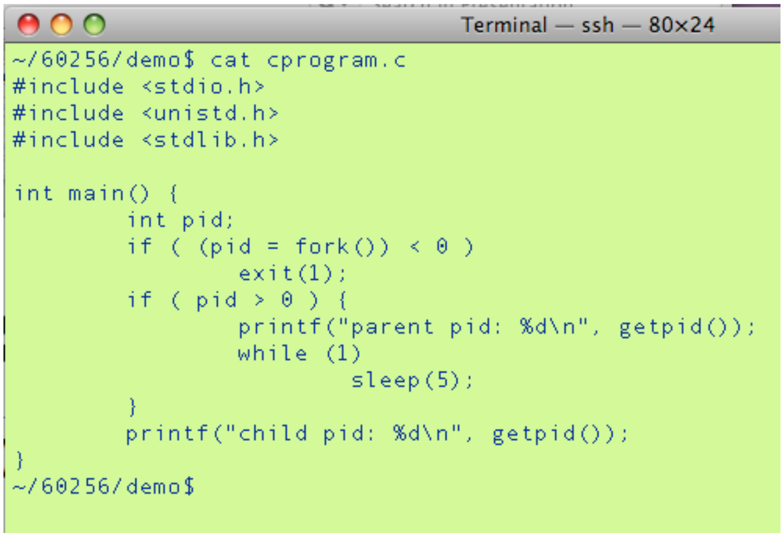
```
~/60256/demo$ cat cprogram.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
        pid_t pid;
        if (( pid = fork()) < 0 ) {
                perror("fork");
                exit(1);
        }
        if ( pid > 0 ) {
                sleep(1);
                exit(0);
        }
        printf("parent pid is: %d\n", getppid());
        sleep(2);
        printf("parent pid is: %d\n", getppid());
}
~/60256/demo$ a.out
parent pid is: 15178
~/60256/demo$ parent pid is: 1

~/60256/demo$
```

Zombie process

- zombies
  - compared to normal processes, they lose their resources e.g.
    - data
    - code
    - stack
  - however, they remain in system’s process table waiting for acceptance of their return

    (system’s process table has a fixed size)

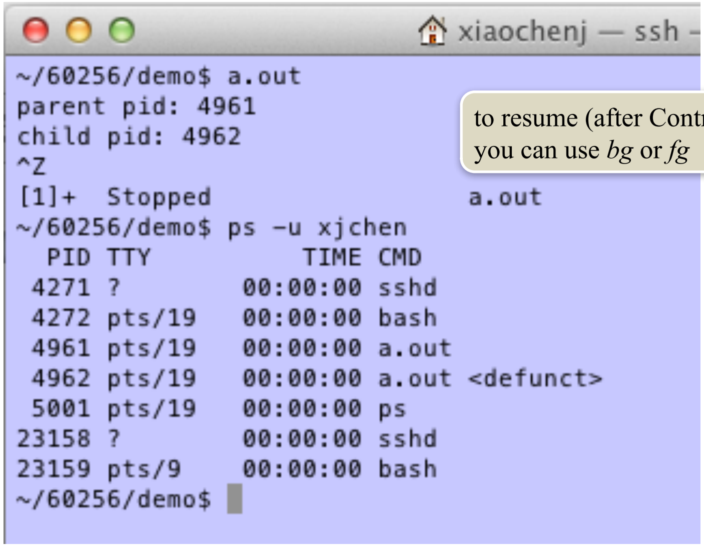
Example: making a zombie process



```
~/60256/demo$ cat cprogram.c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
        int pid;
        if ( (pid = fork()) < 0 )
                exit(1);
        if ( pid > 0 ) {
                printf("parent pid: %d\n", getpid());
                while (1)
                        sleep(5);
        }
        printf("child pid: %d\n", getpid());
}
~/60256/demo$
```

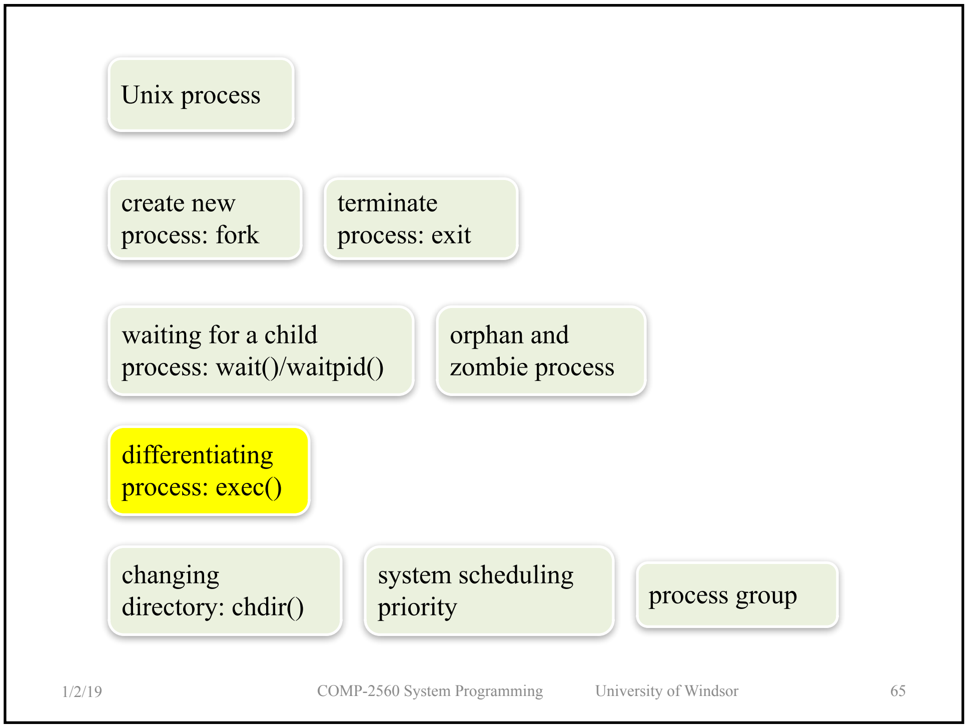
command to check the running processes

> ps –u your_user_id



```
~/60256/demo$ a.out
parent pid: 4961
child pid: 4962
^Z
[1]+  Stopped                 a.out
~/60256/demo$ ps -u xjchen
  PID TTY          TIME CMD
 4271 ?        00:00:00 sshd
 4272 pts/19   00:00:00 bash
 4961 pts/19   00:00:00 a.out
 4962 pts/19   00:00:00 a.out <defunct>
 5001 pts/19   00:00:00 ps
23158 ?        00:00:00 sshd
23159 pts/9    00:00:00 bash
~/60256/demo$
```

to resume (after Control-Z), you can use *bg* or *fg*



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

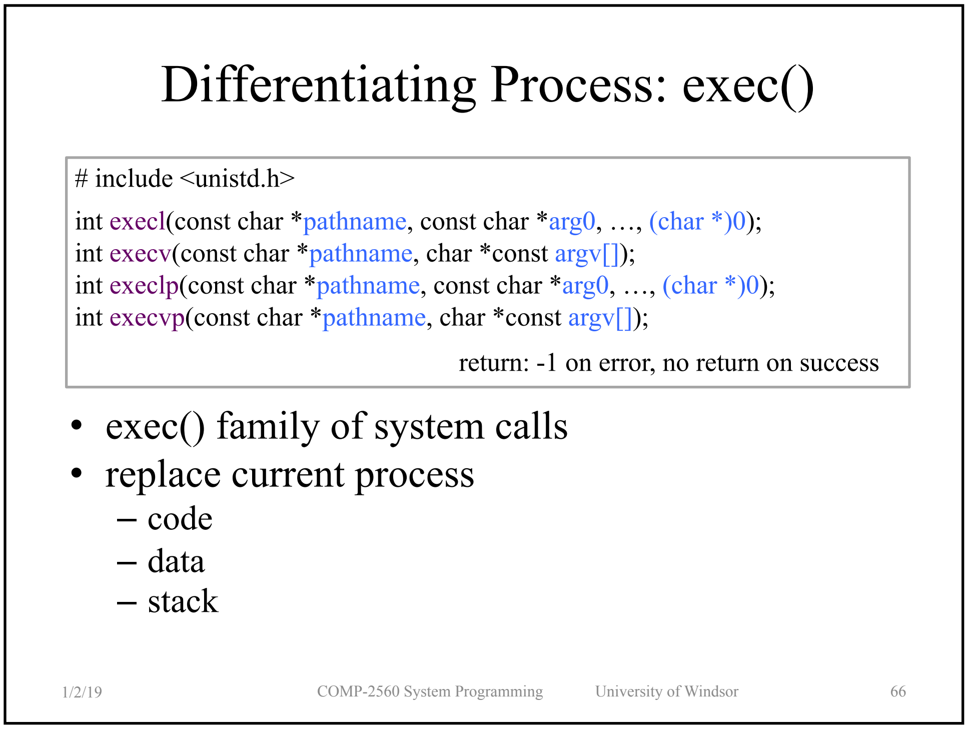
orphan and zombie process

differentiating process: exec()

changing directory: chdir()

system scheduling priority

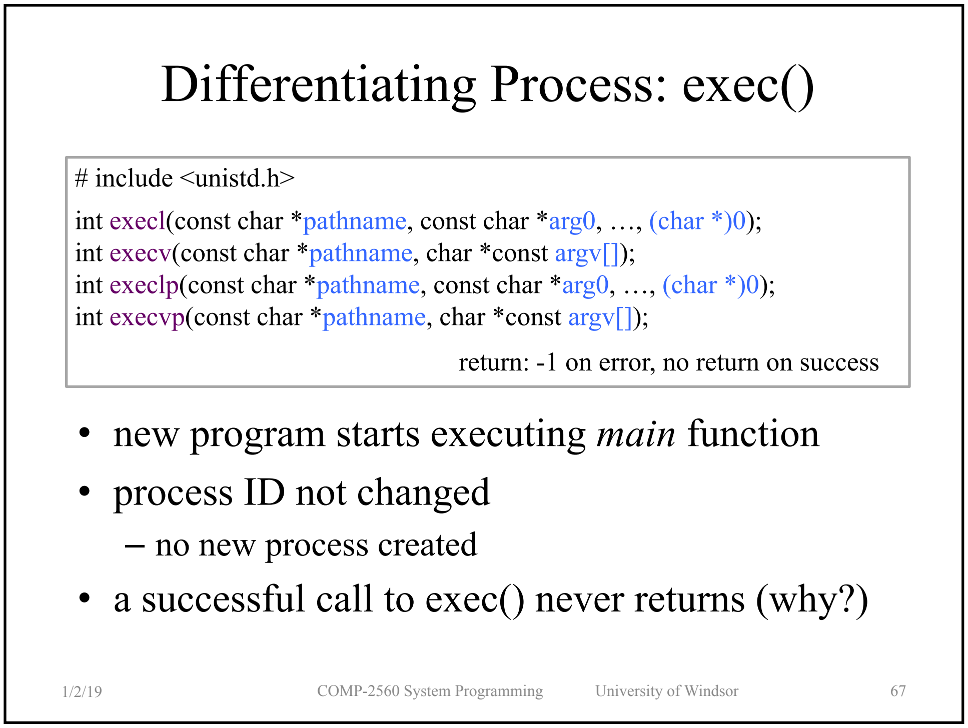
process group



Differentiating Process: exec()

```
# include <unistd.h>
int execl(const char *pathname, const char *arg0, …, (char *)0);
int execv(const char *pathname, char *const argv[]);
int execlp(const char *pathname, const char *arg0, …, (char *)0);
int execvp(const char *pathname, char *const argv[]);
                                        return: -1 on error, no return on success
```

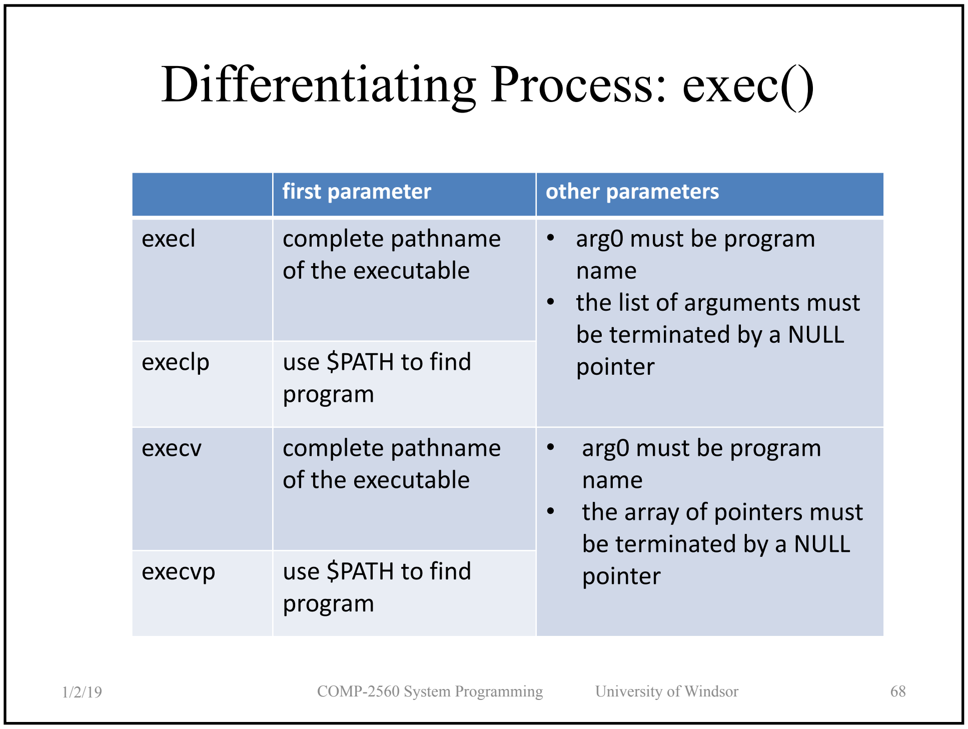
- exec() family of system calls
- replace current process
  - code
  - data
  - stack



Differentiating Process: exec()

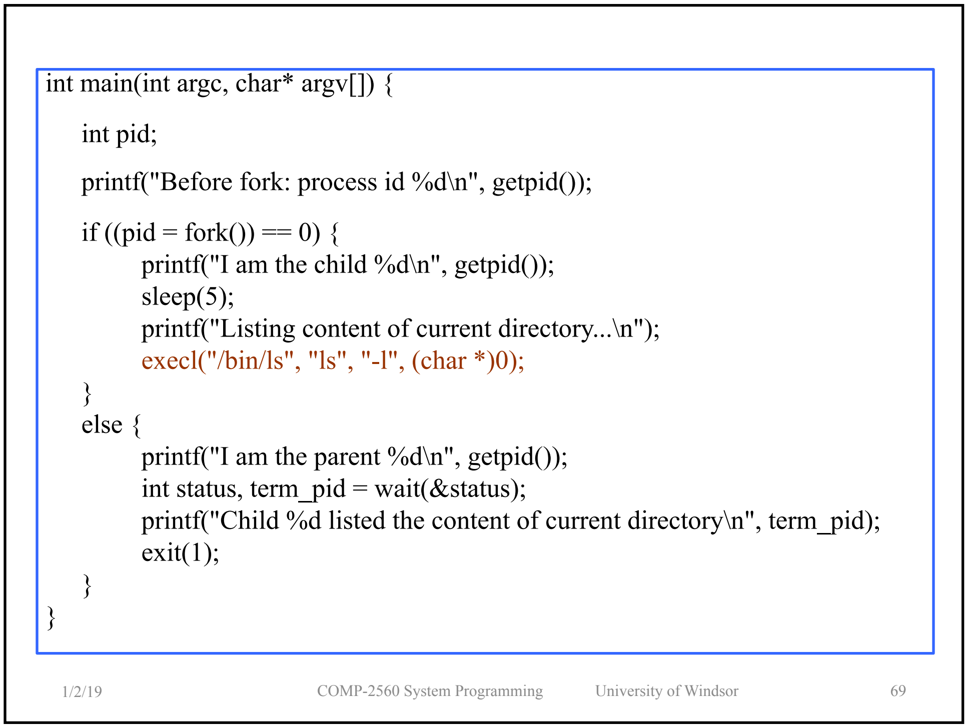
```
# include <unistd.h>
int execl(const char *pathname, const char *arg0, …, (char *)0);
int execv(const char *pathname, char *const argv[]);
int execlp(const char *pathname, const char *arg0, …, (char *)0);
int execvp(const char *pathname, char *const argv[]);
                                    return: -1 on error, no return on success
```

- new program starts executing *main* function
- process ID not changed
  - no new process created
- a successful call to exec() never returns (why?)

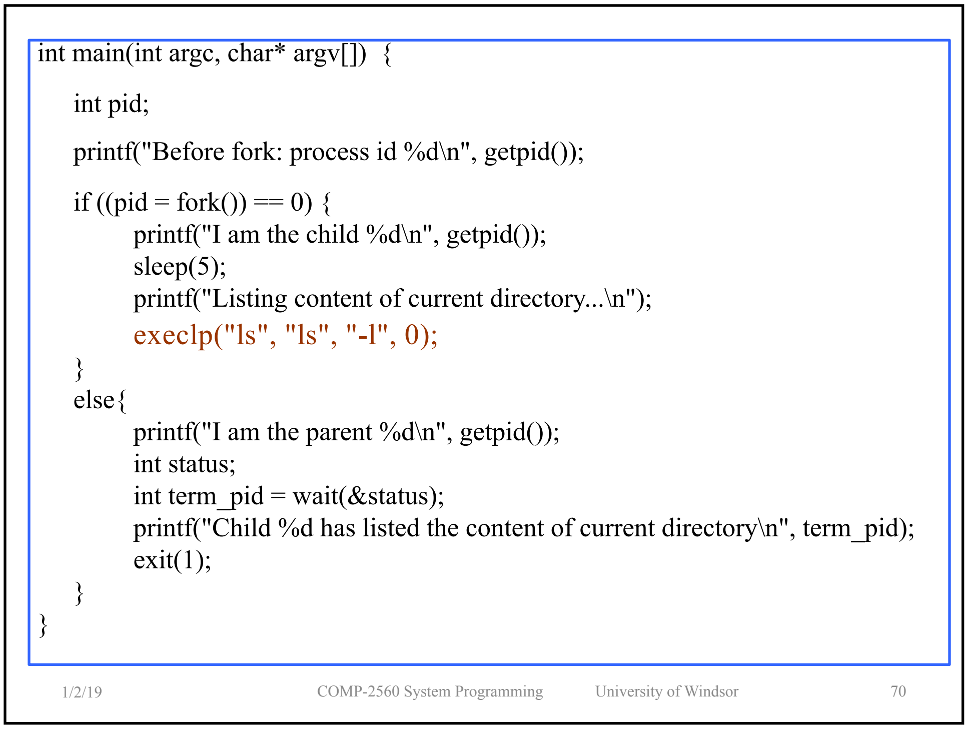


Differentiating Process: exec()

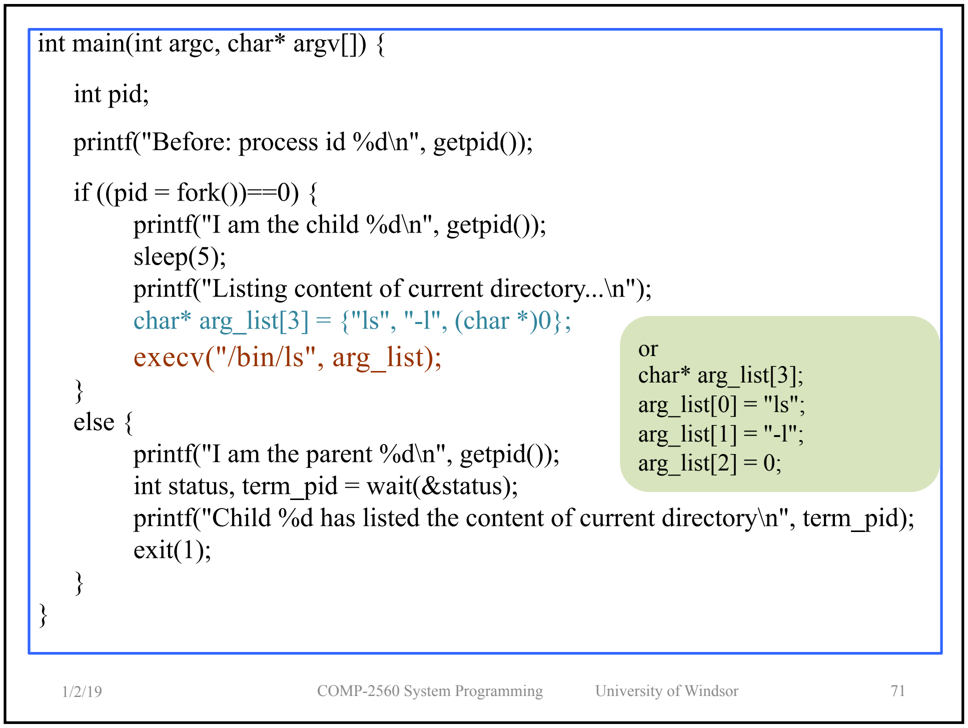
	first parameter	other parameters
execl	complete pathname of the executable	• arg0 must be program name • the list of arguments must be terminated by a NULL pointer
execlp	use $PATH to find program	
execv	complete pathname of the executable	• arg0 must be program name • the array of pointers must be terminated by a NULL pointer
execvp	use $PATH to find program	



```
int main(int argc, char* argv[]) {

    int pid;

    printf("Before fork: process id %d\n", getpid());

    if ((pid = fork()) == 0) {
            printf("I am the child %d\n", getpid());
            sleep(5);
            printf("Listing content of current directory...\n");
            execl("/bin/ls", "ls", "-l", (char *)0);
    }
    else {
            printf("I am the parent %d\n", getpid());
            int status, term_pid = wait(&status);
            printf("Child %d listed the content of current directory\n", term_pid);
            exit(1);
    }
}
```

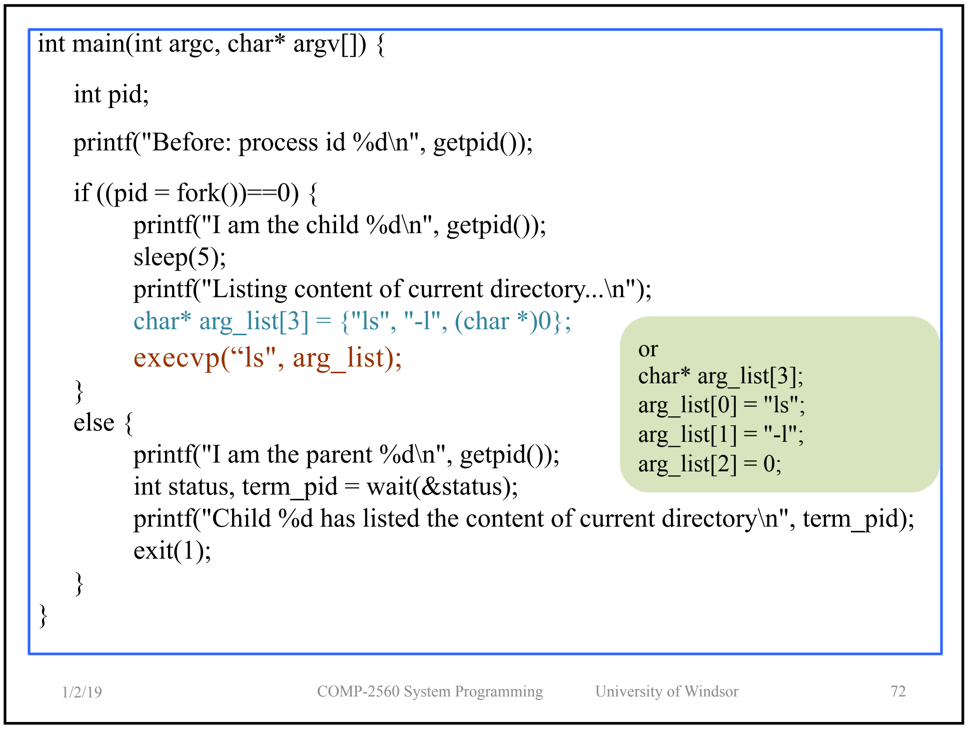


```
int main(int argc, char* argv[])  {

    int pid;

    printf("Before fork: process id %d\n", getpid());

    if ((pid = fork()) == 0) {
            printf("I am the child %d\n", getpid());
            sleep(5);
            printf("Listing content of current directory...\n");
            execlp("ls", "ls", "-l", 0);
    }
    else{
            printf("I am the parent %d\n", getpid());
            int status;
            int term_pid = wait(&status);
            printf("Child %d has listed the content of current directory\n", term_pid);
            exit(1);
    }
}
```



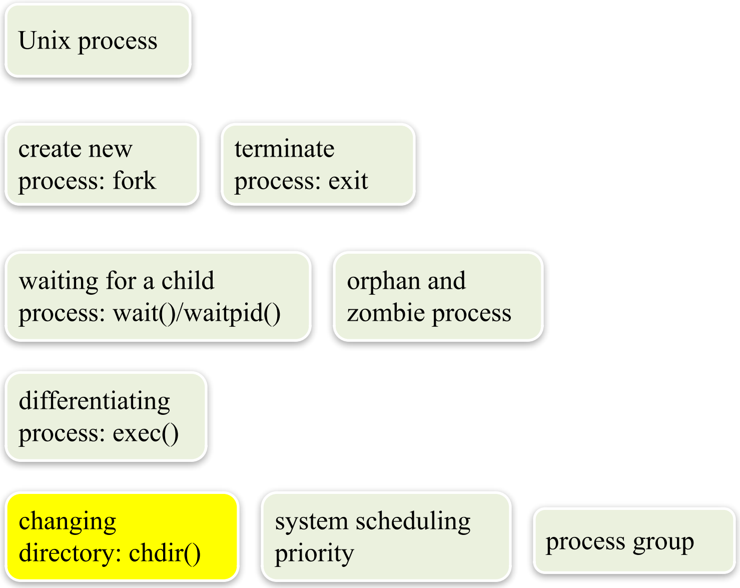
```
int main(int argc, char* argv[]) {

    int pid;

    printf("Before: process id %d\n", getpid());

    if ((pid = fork())==0) {
            printf("I am the child %d\n", getpid());
            sleep(5);
            printf("Listing content of current directory...\n");
            char* arg_list[3] = {"ls", "-l", (char *)0};
            execv("/bin/ls", arg_list);
    }
    else {
            printf("I am the parent %d\n", getpid());
            int status, term_pid = wait(&status);
            printf("Child %d has listed the content of current directory\n", term_pid);
            exit(1);
    }
}
```

```
or
char* arg_list[3];
arg_list[0] = "ls";
arg_list[1] = "-l";
arg_list[2] = 0;
```



```
int main(int argc, char* argv[]) {

    int pid;

    printf("Before: process id %d\n", getpid());

    if ((pid = fork())==0) {
            printf("I am the child %d\n", getpid());
            sleep(5);
            printf("Listing content of current directory...\n");
            char* arg_list[3] = {"ls", "-l", (char *)0};
            execvp(“ls", arg_list);
    }
    else {
            printf("I am the parent %d\n", getpid());
            int status, term_pid = wait(&status);
            printf("Child %d has listed the content of current directory\n", term_pid);
            exit(1);
    }
}
```

```
or
char* arg_list[3];
arg_list[0] = "ls";
arg_list[1] = "-l";
arg_list[2] = 0;
```



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

orphan and zombie process

differentiating process: exec()

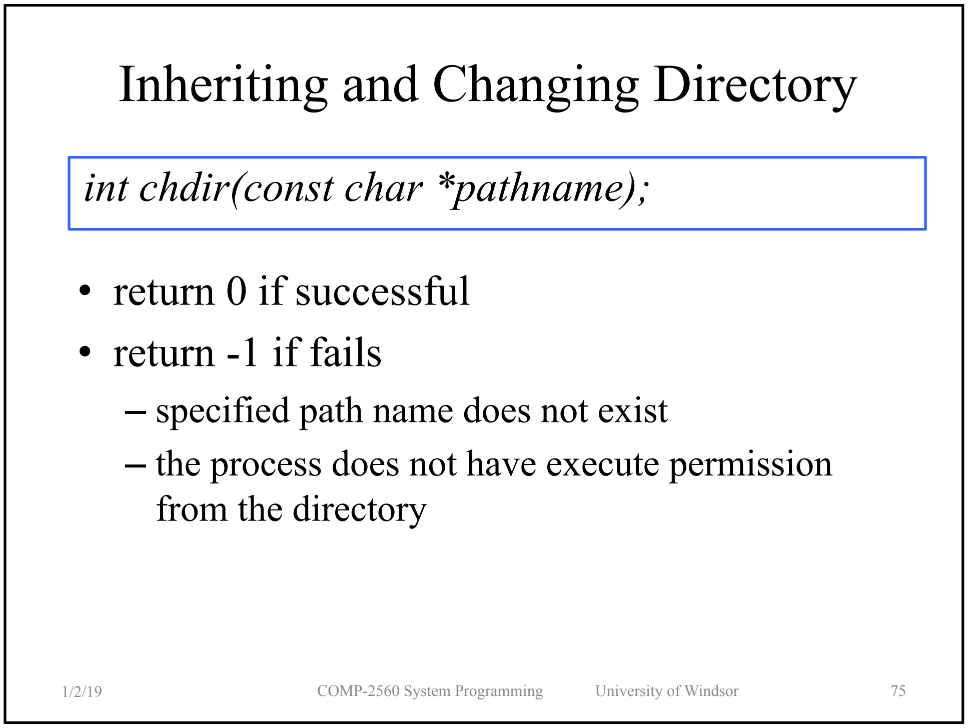
changing directory: chdir()

system scheduling priority

process group

Inheriting and Changing Directory

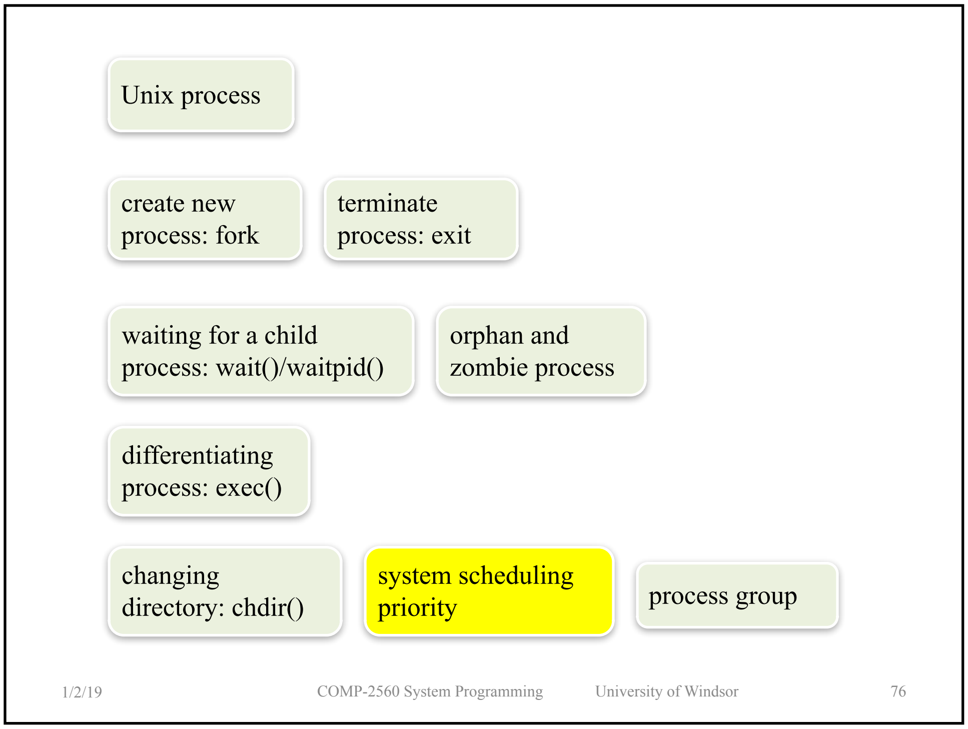
- child process inherits current working directory of parent
- child process can change working directory using *chdir()*



Inheriting and Changing Directory

*int chdir(const char *pathname);*

- return 0 if successful
- return -1 if fails
  - specified path name does not exist
  - the process does not have execute permission from the directory



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

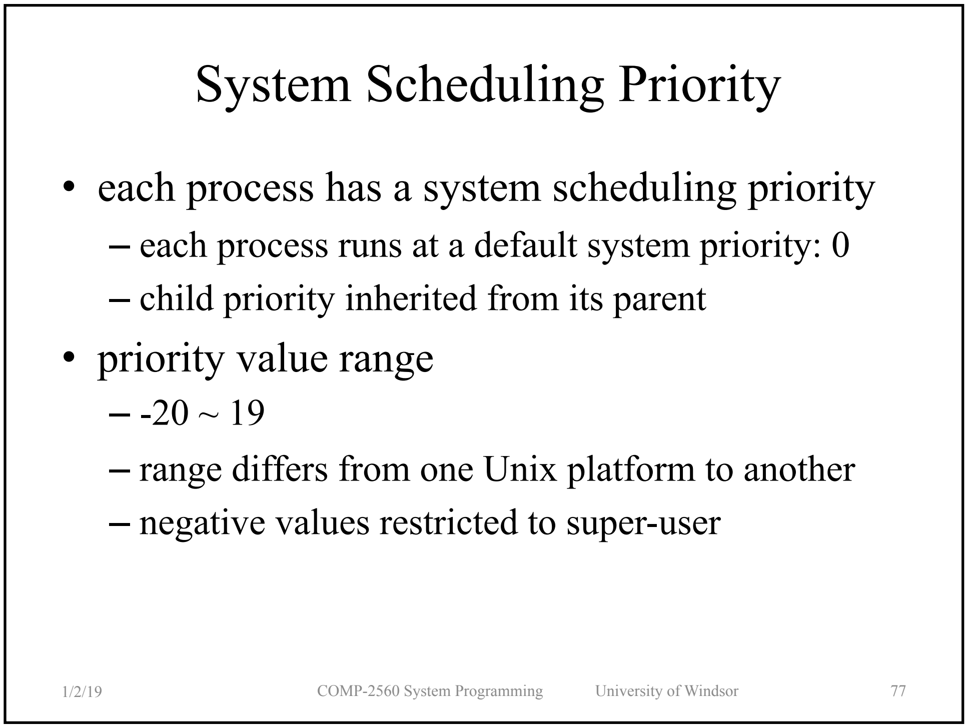
orphan and zombie process

differentiating process: exec()

changing directory: chdir()

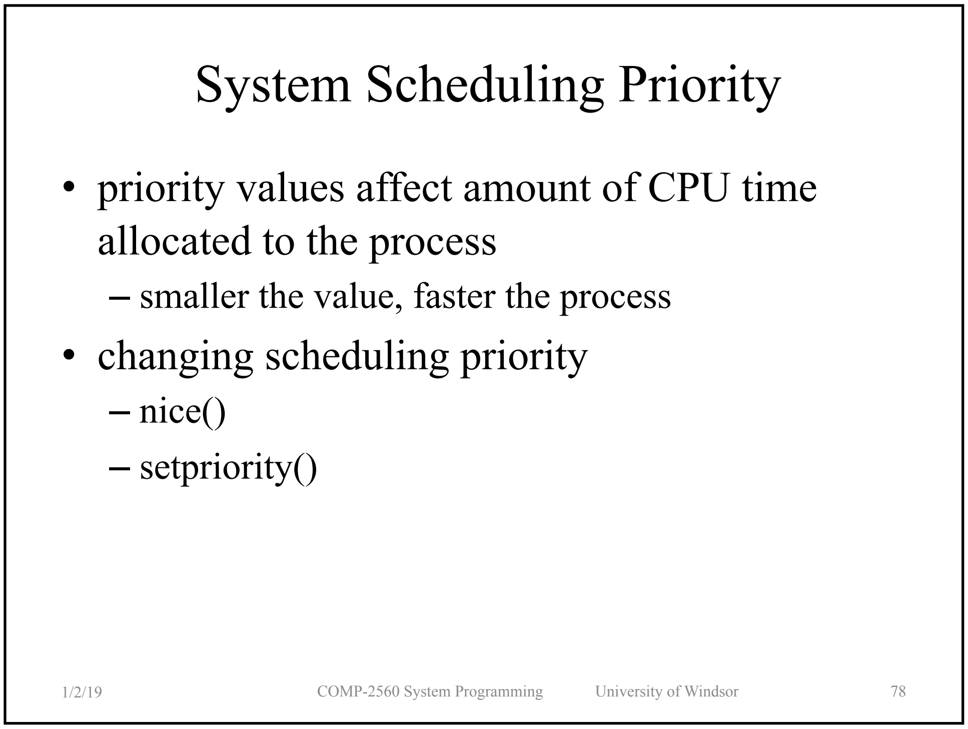
system scheduling priority

process group



System Scheduling Priority

- each process has a system scheduling priority
  - each process runs at a default system priority: 0
  - child priority inherited from its parent
- priority value range
  - -20 ~ 19
  - range differs from one Unix platform to another
  - negative values restricted to super-user



System Scheduling Priority

- priority values affect amount of CPU time allocated to the process
  - smaller the value, faster the process
- changing scheduling priority
  - nice()
  - setpriority()

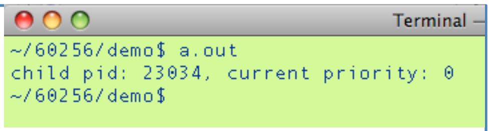
getpriority() and setpriority()

```
<sys/resource.h>

int getpriority(int which, id_t who)
                              returns priority value on success; -1 on failure
```

- obtains current scheduling priority of a process, process group, or user
- *which*
  - identifies whether it is a process (PRIO_PROCESS), process group, etc.
- *who*
  - process id, group id etc
  - if *who* is 0, the calling process (or group etc.) is considered

Example: getpriority()

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

int main() {
        int pid, status;
        if ( (pid = fork()) < 0 )
                exit(1);
        if ( pid == 0 )
                printf("child pid: %d, current priority: %d\n",
                        getpid(), getpriority(PRIO_PROCESS, getpid()));

        else
                wait(&status);
}
```



```
~/60256/demo$ a.out
child pid: 23034, current priority: 0
~/60256/demo$
```

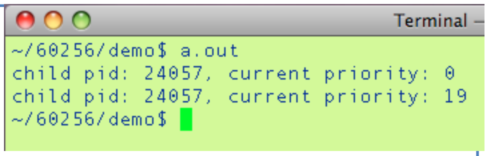
System Scheduling Priority

```
int nice(int delta)
```

- adds delta to current value
- only super-user processes can have a negative value
- returns new priority value if successful; -1 o.w.

example: nice()

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

int main() {
        int pid, status;
        if ( (pid = fork()) < 0 )
                exit(1);
        if ( pid == 0 ) {
                printf("child pid: %d, current priority: %d\n",
                        getpid(), getpriority(PRIO_PROCESS, getpid()));
                nice(19);
                printf("child pid: %d, current priority: %d\n",
                        getpid(), getpriority(PRIO_PROCESS, getpid()));
        }
        else
                wait(&status);
}
```



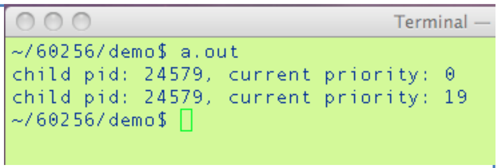
```
~/60256/demo$ a.out
child pid: 24057, current priority: 0
child pid: 24057, current priority: 19
~/60256/demo$
```

getpriority() and setpriority()

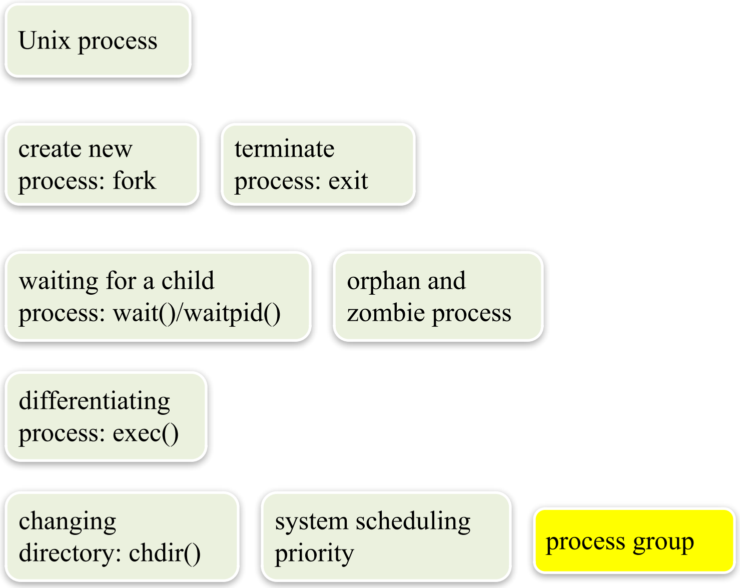
```
<sys/resource.h>
int setpriority(int which, id_t who, int priority)
                    returns 0 on success; -1 on failure
```

example: parent sets priority of the child

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
int main() {
        int pid, status;
        if ( (pid = fork()) < 0 )
                exit(1);
        if ( pid == 0 ) {
                printf("child pid: %d, current priority: %d\n",
                        getpid(), getpriority(PRIO_PROCESS, getpid()));
                sleep(5);
                printf("child pid: %d, current priority: %d\n",
                        getpid(), getpriority(PRIO_PROCESS, getpid()));
        }
        else {
                sleep(1);
                setpriority(PRIO_PROCESS, pid, 19);
                wait(&status);
        }
}
```



```
~/60256/demo$ a.out
child pid: 24579, current priority: 0
child pid: 24579, current priority: 19
~/60256/demo$ 
```



Unix process

create new process: fork

terminate process: exit

waiting for a child process: wait()/waitpid()

orphan and zombie process

differentiating process: exec()

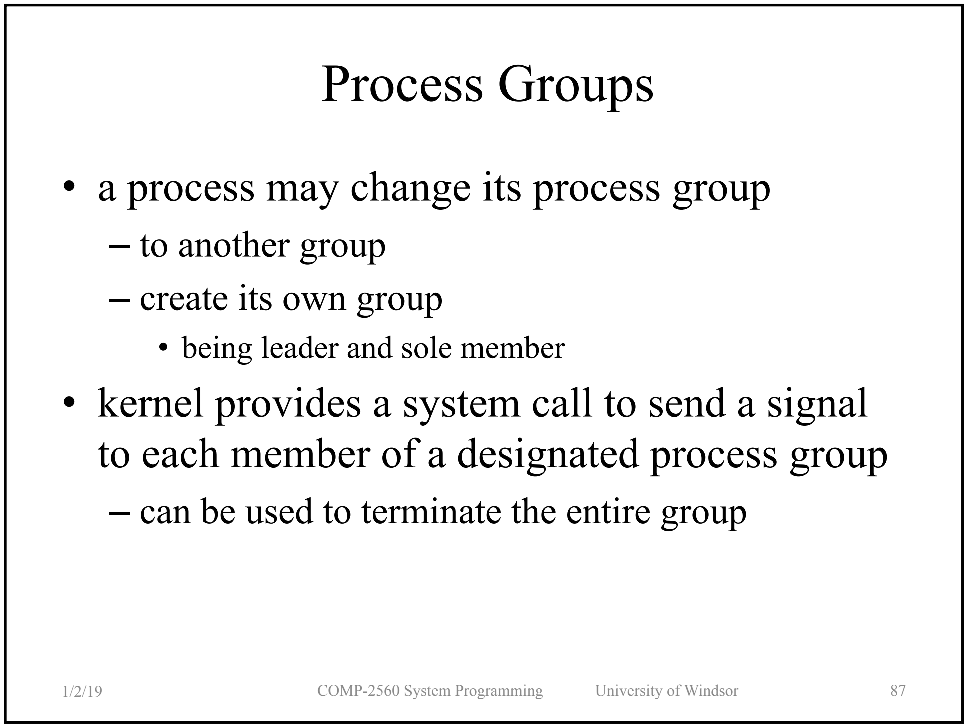
changing directory: chdir()

system scheduling priority

process group

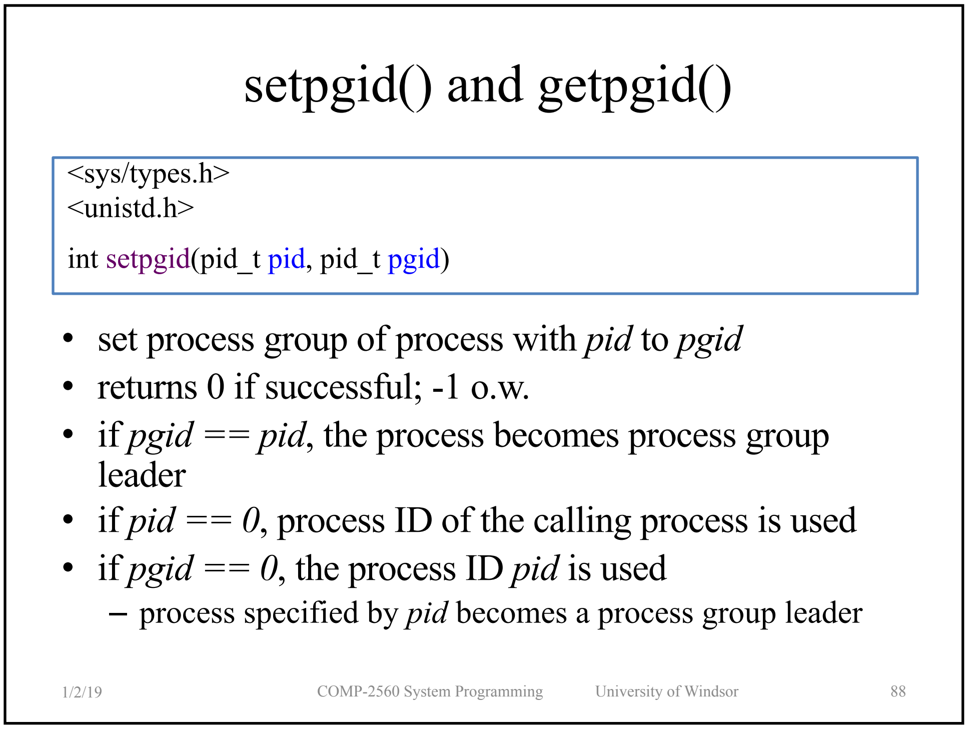
Process Groups

- every process is a member of a process group
  - a child inherits process group from parent
  - when calling exec(), process group remains the same
- one of group members is the group leader
  - each group member has the group leader’s process ID and its process-group-ID



Process Groups

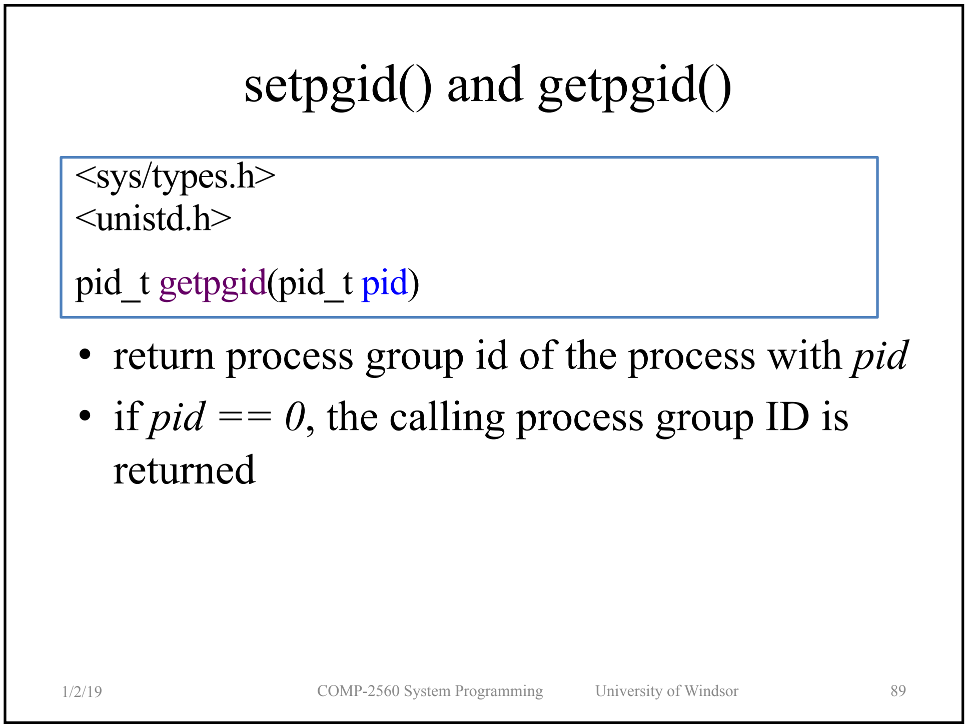
- a process may change its process group
  - to another group
  - create its own group
    - being leader and sole member
- kernel provides a system call to send a signal to each member of a designated process group
  - can be used to terminate the entire group



setpgid() and getpgid()

```
<sys/types.h>
<unistd.h>

int setpgid(pid_t pid, pid_t pgid)
```

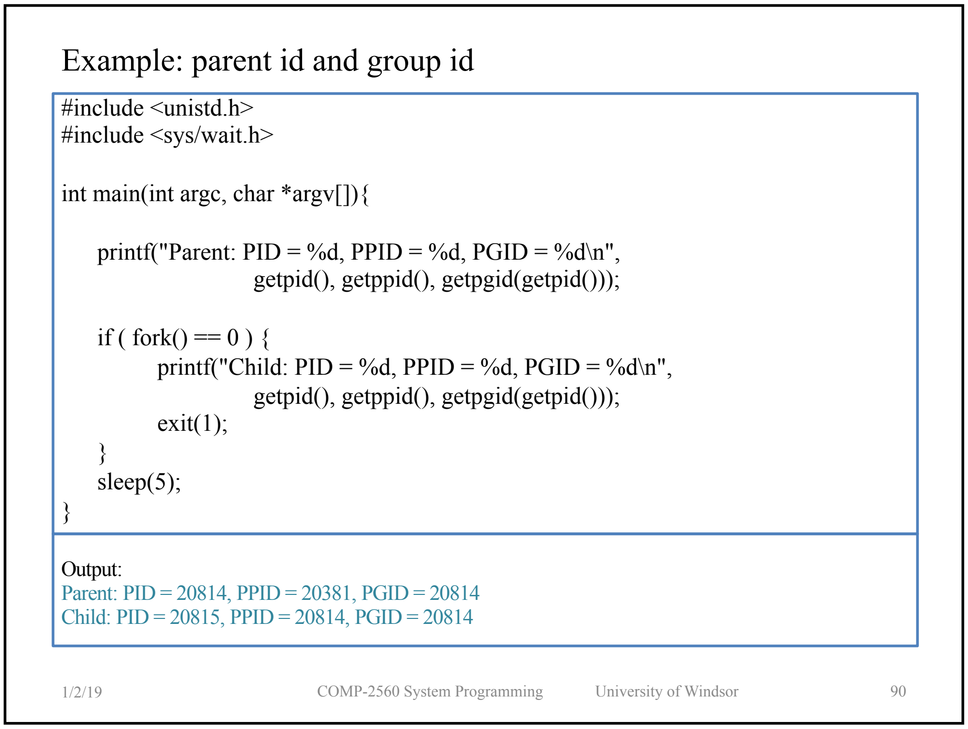
- set process group of process with *pid* to *pgid*
- returns 0 if successful; -1 o.w.
- if *pgid == pid*, the process becomes process group leader
- if *pid == 0*, process ID of the calling process is used
- if *pgid == 0*, the process ID *pid* is used
  - process specified by *pid* becomes a process group leader



setpgid() and getpgid()

```
<sys/types.h>
<unistd.h>

pid_t getpgid(pid_t pid)
```

- return process group id of the process with *pid*
- if *pid == 0*, the calling process group ID is returned

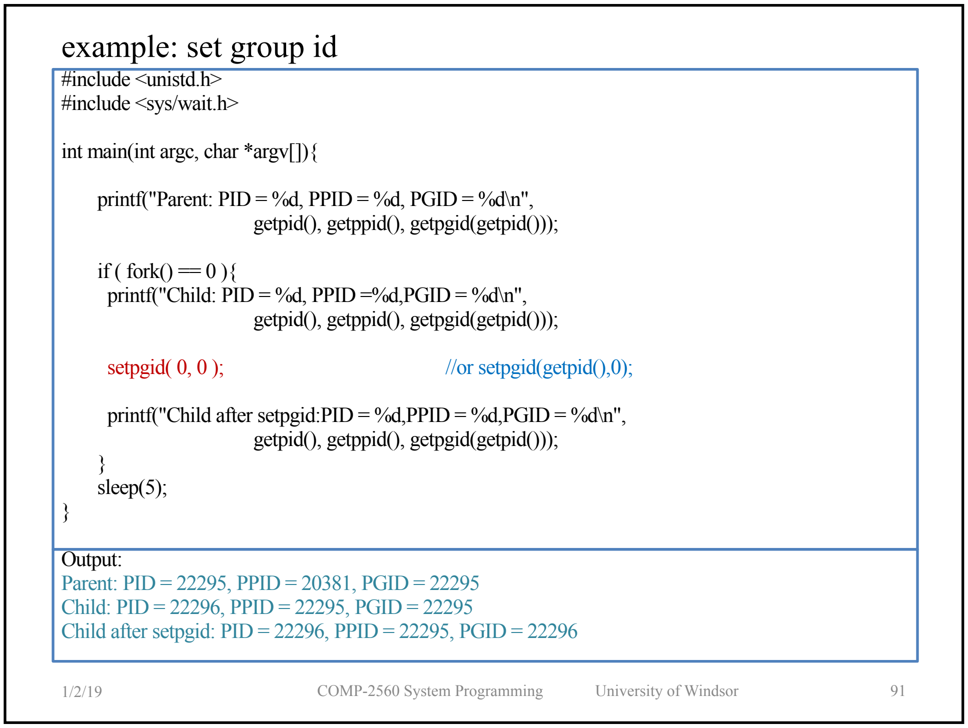


Example: parent id and group id

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

int main(int argc, char *argv[]){

    printf("Parent: PID = %d, PPID = %d, PGID = %d\n",
                        getpid(), getppid(), getpgid(getpid()));

    if ( fork() == 0 ) {
            printf("Child: PID = %d, PPID = %d, PGID = %d\n",
                        getpid(), getppid(), getpgid(getpid()));
            exit(1);
    }
    sleep(5);
}
```

Output:

```
Parent: PID = 20814, PPID = 20381, PGID = 20814
Child: PID = 20815, PPID = 20814, PGID = 20814
```



example: set group id

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

int main(int argc, char *argv[]){

     printf("Parent: PID = %d, PPID = %d, PGID = %d\n",
                         getpid(), getppid(), getpgid(getpid()));

     if ( fork() == 0 ){
       printf("Child: PID = %d, PPID =%d,PGID = %d\n",
                         getpid(), getppid(), getpgid(getpid()));

       setpgid( 0, 0 );                    //or setpgid(getpid(),0);

       printf("Child after setpgid:PID = %d,PPID = %d,PGID = %d\n",
                         getpid(), getppid(), getpgid(getpid()));
     }
     sleep(5);
}
```

Output:

```
Parent: PID = 22295, PPID = 20381, PGID = 22295
Child: PID = 22296, PPID = 22295, PGID = 22295
Child after setpgid: PID = 22296, PPID = 22295, PGID = 22296
```