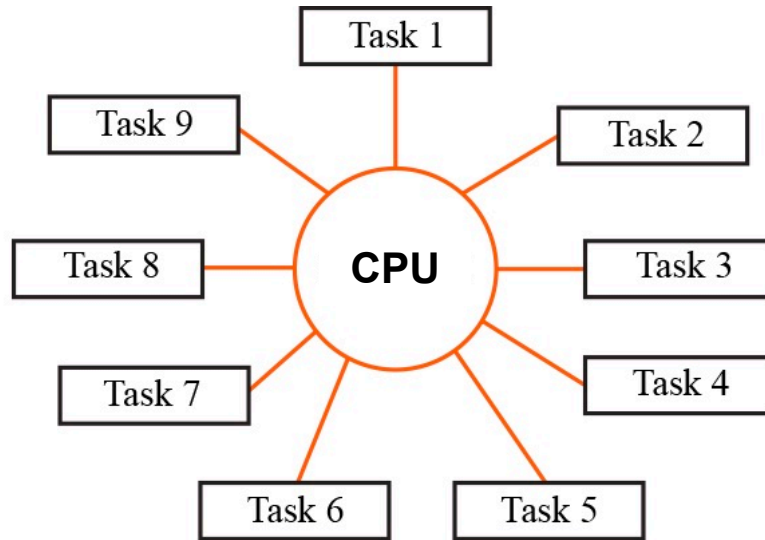
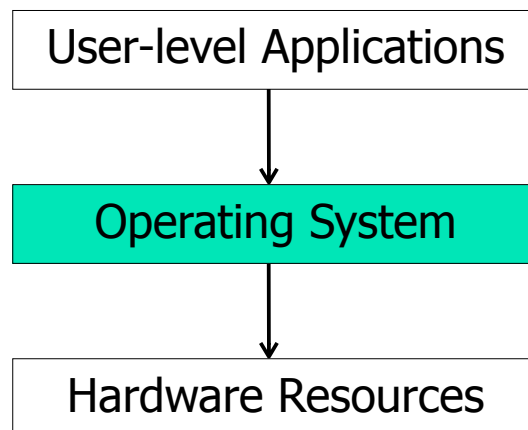


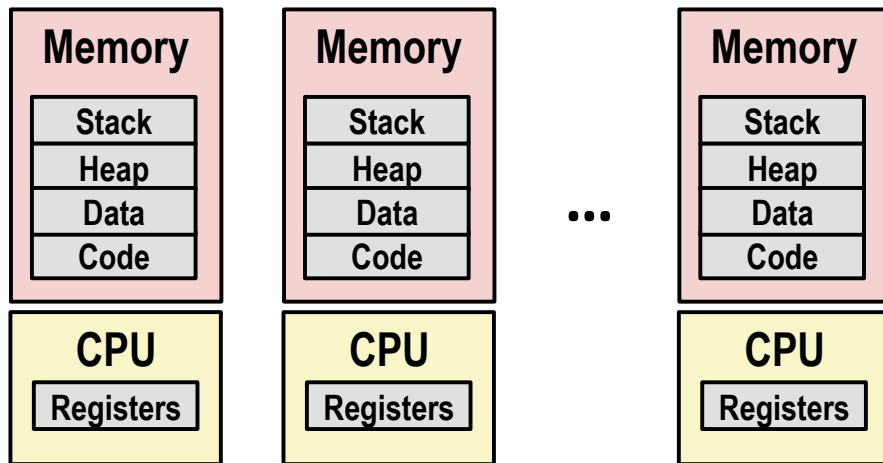
Global Control Flow



Operating System

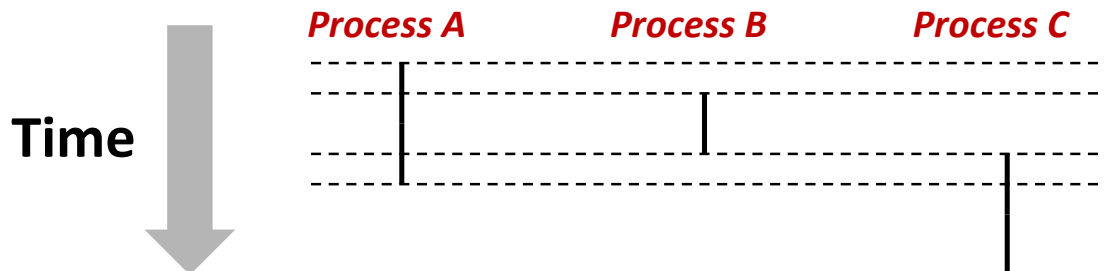


Processes



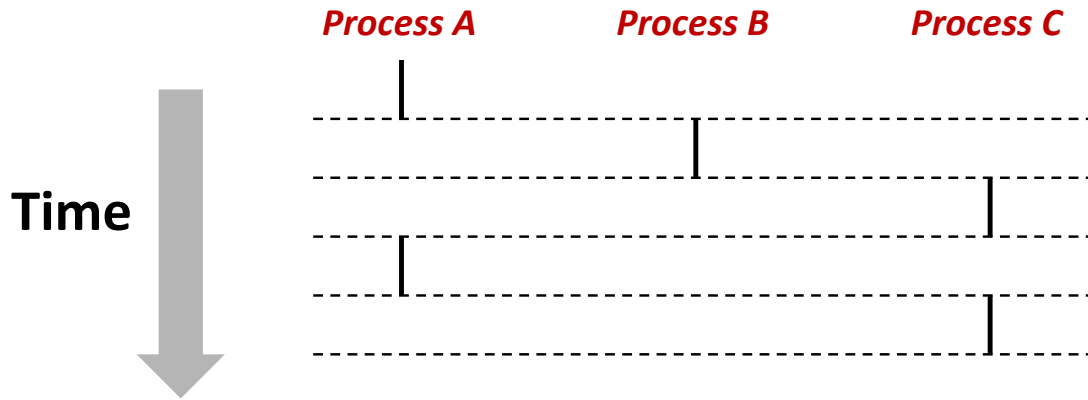
Control Flow Abstraction

Simultaneous execution

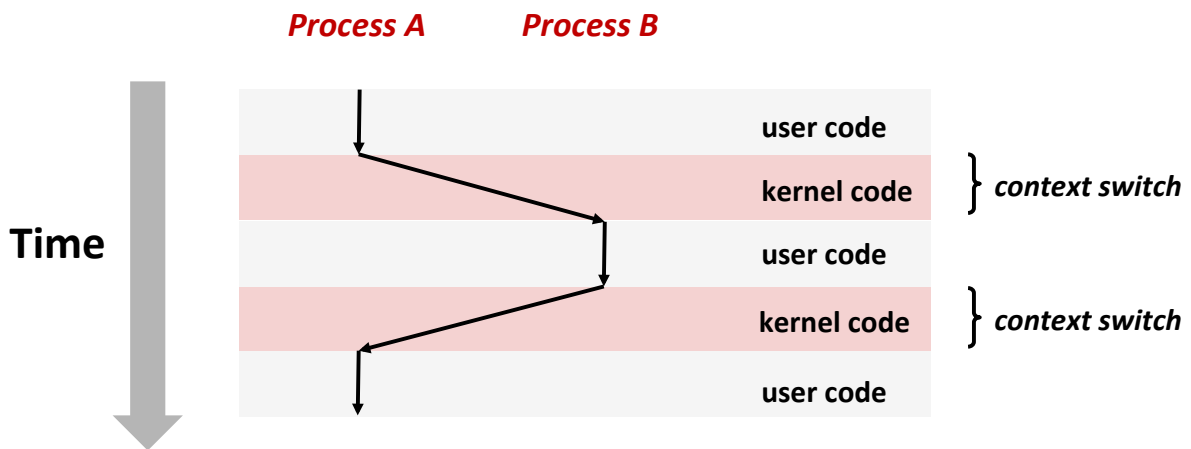


Control Flow Reality

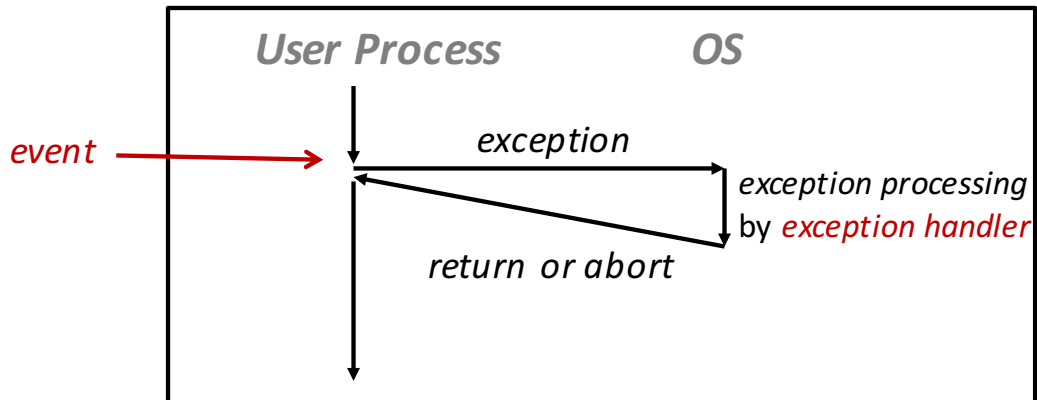
Time-sharing!



Context Switching



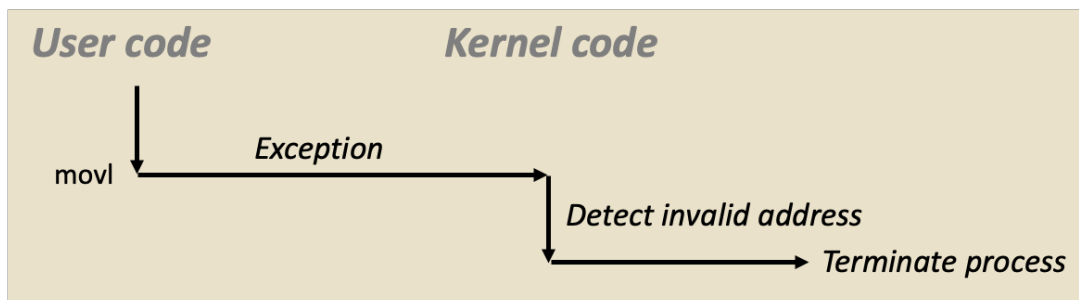
Exceptional Control Flow



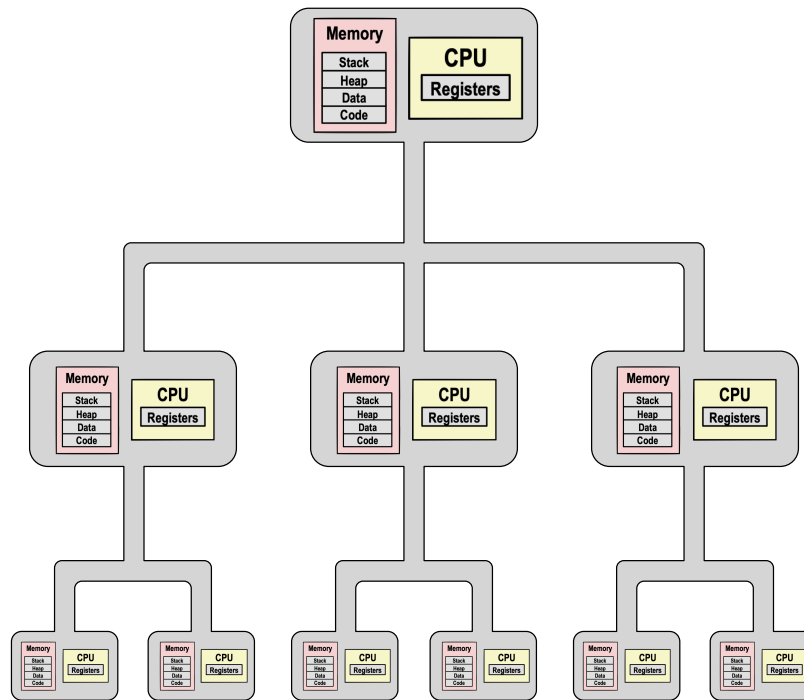
Example: Segmentation Fault

```
int a[1000];  
  
int main() {  
    a[5000] = 7;  
    ...  
}
```

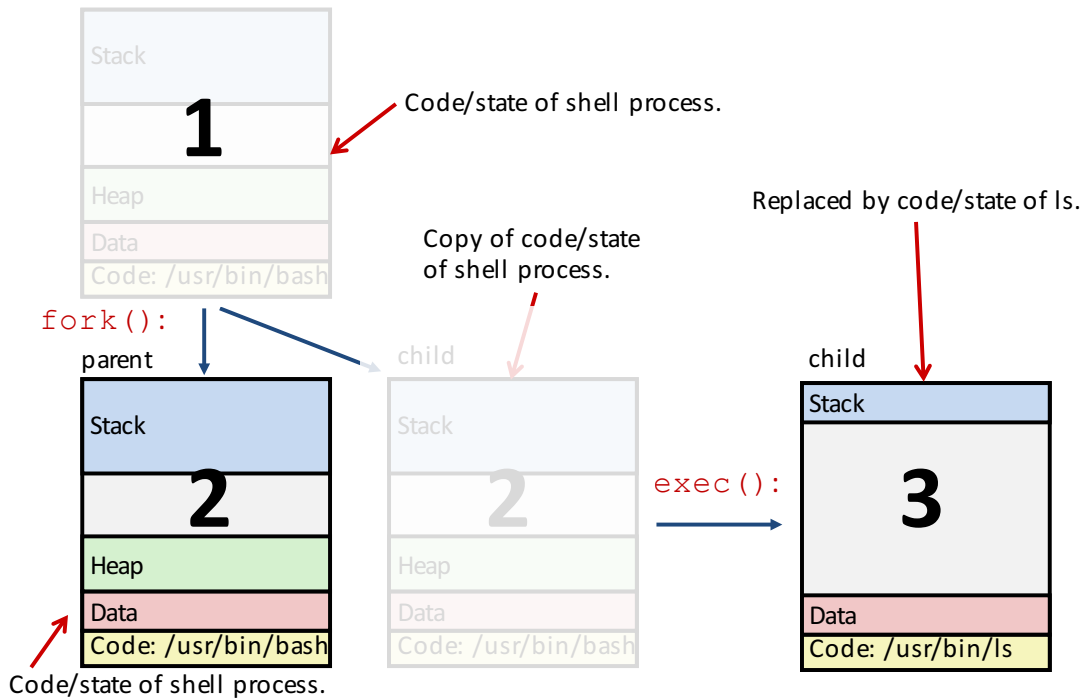
```
80483b7: c7 05 60 e3 04 08 07 movl $0x7,0x804e360
```



Process Management



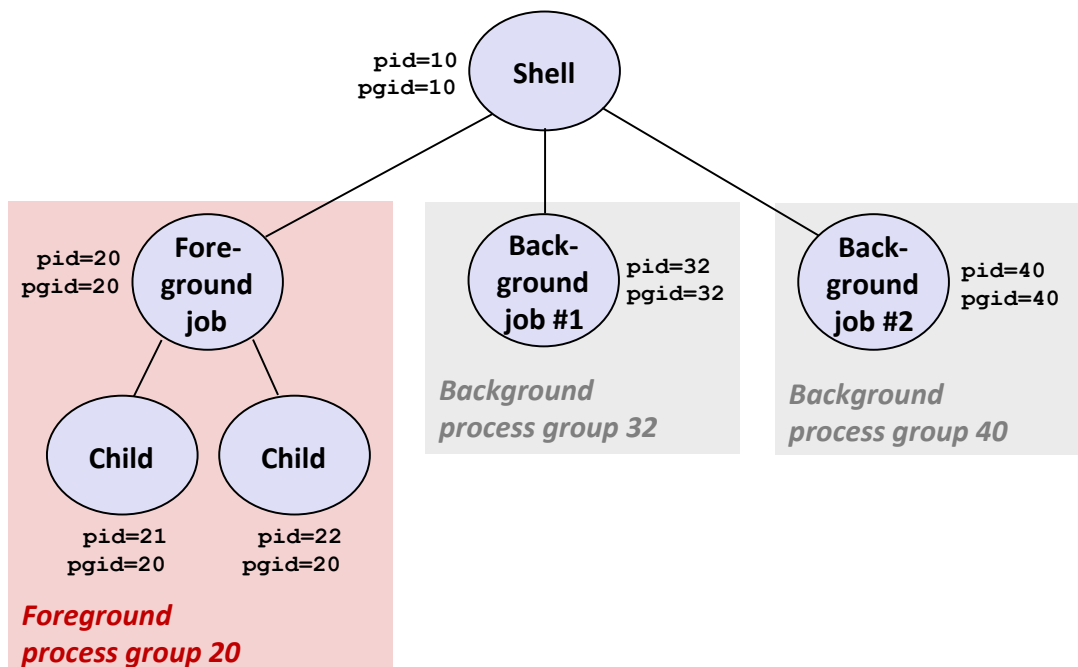
Fork & Exec Example



Basic Shell Design

```
while (true) {  
    Print command prompt.  
    Read command line from user.  
    Parse command line.  
    If command is built-in, execute it.  
    Else, fork process  
        in child:  
            Execute requested command with exec  
                (never returns)  
        in parent:  
            Wait for child to complete with waitpid  
}
```

Process Groups



Signals

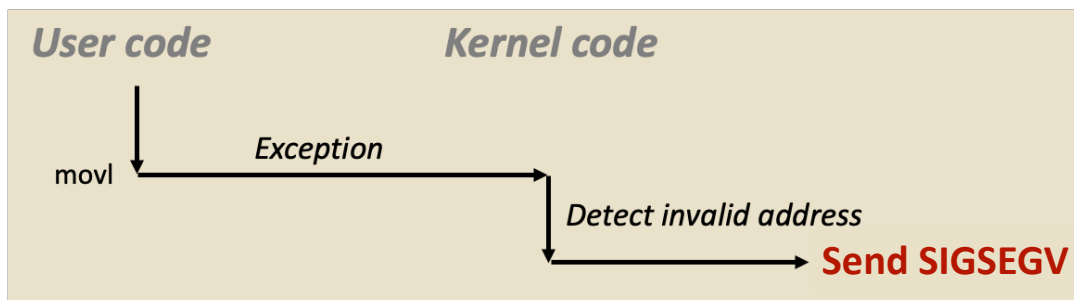
<i>ID</i>	<i>Name</i>	<i>Signal Description</i>	<i>Shell Shortcut</i>	<i>Default Action</i>	<i>Override?</i>
2	SIGINT	Interrupt process	Control-C	Terminate	Yes
9	SIGKILL	Kill process (immediately)		Terminate	No
11	SIGSEGV	Segmentation fault		Terminate	Yes
15	SIGTERM	Kill process (politely)		Terminate	Yes
17	SIGCHLD	Child stopped or terminated		Ignore	Yes
18	SIGCONT	Continue stopped process		Continue (Resume)	No
19	SIGSTOP	Stop process (immediately)		Stop (Suspend)	No
20	SIGTSTP	Stop process (politely)	Control-Z	Stop (Suspend)	Yes

Segmentation Fault (redux)

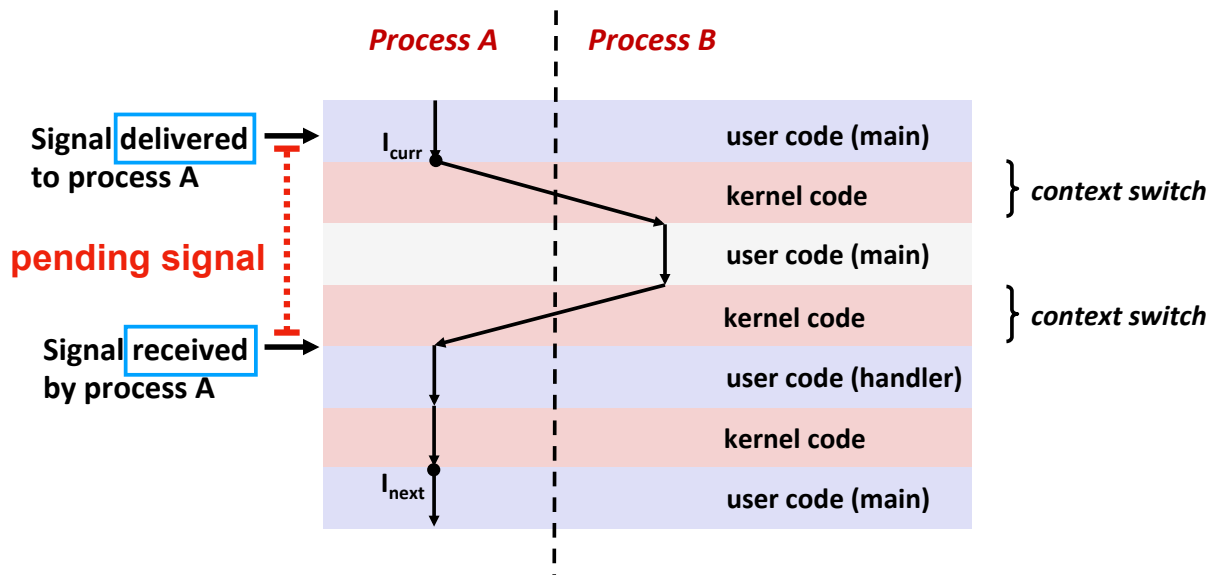
```
int a[1000];

int main() {
    a[5000] = 7;
    ...
}
```

```
80483b7: c7 05 60 e3 04 08 07 movl $0x7,0x804e360
```



Signal Handler Control Flow



Zombies!



Reaping Zombies

```
pid_t waitpid(pid_t pid, int* stat, int ops)
```

wait set



Option Macros

```
pid_t waitpid(pid_t pid, int* stat, int ops)
```

WUNTRACED

also wait for stopped
(suspended) children

WNOHANG

return immediately if child not
already terminated/stopped

Status Macros

```
pid_t waitpid(pid_t pid, int* stat, int ops)
```

WEXITSTATUS(*stat)

numeric **exit code** of child

WIFEXITED(*stat)

true if child **terminated normally**
(called `exit` or returned from `main`)

WIFSIGNALED(*stat)

true if child **terminated by signal**

WIFSTOPPED(*stat)

true if child **stopped** (suspended) by signal

Basic Shell Design (redux)

```
while (true) {  
    Print command prompt.  
    Read command line from user.  
    Parse command line.  
    If command is built-in, execute it.  
    Else, fork process  
        in child:  
            Execute requested command with exec  
            (never returns)  
        in parent:  
            Wait for child to complete with waitpid  
}
```

How to reap background jobs?

Signals (redux)

<i>ID</i>	<i>Name</i>	<i>Signal Description</i>	<i>Shell Shortcut</i>	<i>Default Action</i>	<i>Override?</i>
2	SIGINT	Interrupt process	Control-C	Terminate	Yes
9	SIGKILL	Kill process (immediately)		Terminate	No
11	SIGSEGV	Segmentation fault		Terminate	Yes
15	SIGTERM	Kill process (politely)		Terminate	Yes
17	SIGCHLD	Child stopped or terminated		Ignore	Yes
18	SIGCONT	Continue stopped process		Continue (Resume)	No
19	SIGSTOP	Stop process (immediately)		Stop (Suspend)	No
20	SIGTSTP	Stop process (politely)	Control-Z	Stop (Suspend)	Yes

Reaping in Signal Handler

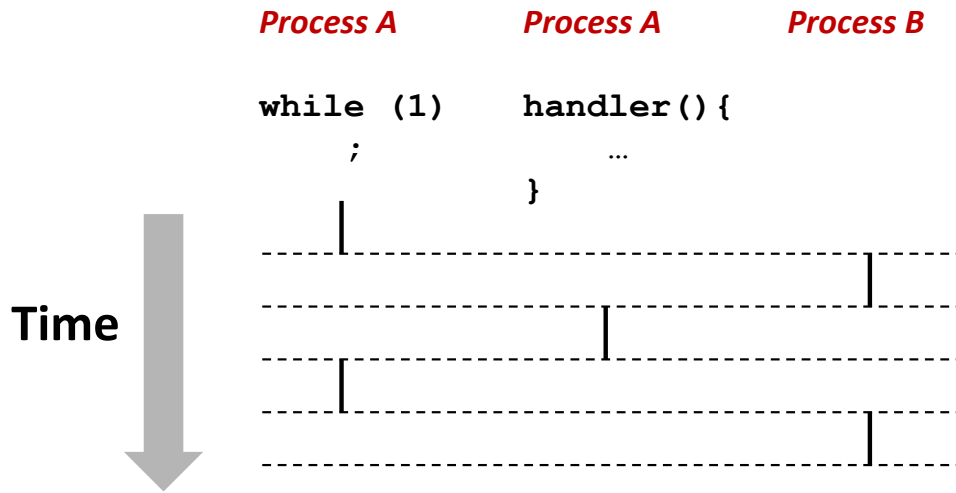
```
int main() {
    pid_t pid;

    Signal(SIGCHLD, sigchld_handler); // install signal handler

    while (1) {
        // print prompt, read cmd from user, etc.
        if ((pid = fork()) == 0) {
            execve(...); // child: run target program
        }
        // parent: wait for child to exit if foreground
    }
    return 0;
}
```

```
void sigchld_handler(int sig) {
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        // reaped child pid
    }
}
```

Signal Handler as Concurrent Flow



Job List Concurrency (1)

```
int main(int argc, char** argv) {  
    pid_t pid;  
  
    Signal(SIGCHLD, sigchld_handler);  
    initjobs(); // initialize job list  
  
    while (1) {  
        if ((pid = fork()) == 0) {  
            execve(...);  
        }  
        addjob(pid); // add child to job list  
    }  
    return 0;  
}
```

Concurrent job
list modification!

```
void sigchld_handler(int sig) {  
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {  
        deletejob(pid); // delete child from job list  
    }  
}
```

Job List Concurrency (2)

```
int main(int argc, char** argv) {
    pid_t pid;
    Signal(SIGCHLD, sigchld_handler);
    initjobs(); // initialize job list
    sigset_t mask; // bit vector
    sigemptyset(&mask); // clear all bits
    sigaddset(&mask, SIGCHLD); // set SIGCHLD bit
    while (1) {
        if ((pid = fork()) == 0) {
            execve(...);
        }
        sigprocmask(SIG_BLOCK, &mask, NULL); // block SIGCHLD
        addjob(pid); // add child to job list
        sigprocmask(SIG_UNBLOCK, &mask, NULL); // unblock SIGCHLD
    }
    return 0;
}
```

Possible delete
before add!

```
void sigchld_handler(int sig) {
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        deletejob(pid); // delete child from job list
    }
}
```

Job List Concurrency (3)

```
int main(int argc, char** argv) {
    pid_t pid;
    Signal(SIGCHLD, sigchld_handler);
    initjobs(); // initialize job list
    sigset_t mask; // bit vector
    sigemptyset(&mask); // clear all bits
    sigaddset(&mask, SIGCHLD); // set SIGCHLD bit
    while (1) {
        sigprocmask(SIG_BLOCK, &mask, NULL); // block SIGCHLD
        if ((pid = fork()) == 0) {
            // unblock in child (inherited from parent)
            sigprocmask(SIG_UNBLOCK, &mask, NULL);
            execve(...);
        }
        addjob(pid); // add child to job list
        sigprocmask(SIG_UNBLOCK, &mask, NULL); // unblock SIGCHLD
    }
    return 0;
}
```

Useful System Calls

fork – Create a new process
execve – Run a new program
kill – Send a signal
waitpid – Wait for and/or reap child process
setpgid – Change process group ID
sigsuspend – Wait until signal received
sigprocmask – Block or unblock signals
sigemptyset – Create empty signal set
sigfillset – Add every signal number to set
sigaddset – Add signal number to set
sigdelset – Delete signal number from set

System Call Error Handling

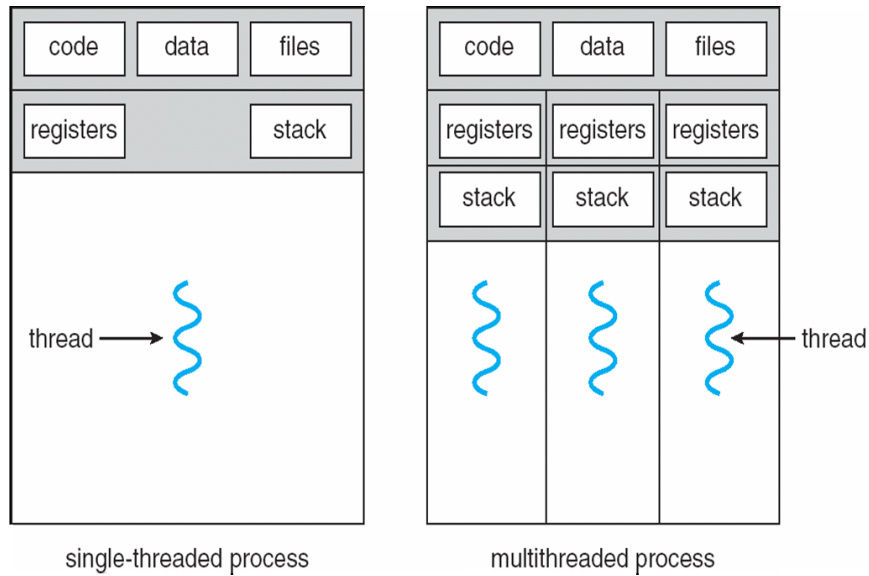
Always check return values!
(<0 means error)

```
pid = fork();  
if (pid < 0) {  
    printf("fork error: %s\n", strerror(errno));  
}
```

global var



Threads



Thread Example

```
/*
 * hello.c - Pthreads "hello, world" program
 */
void* thread(void* vargp);
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}
hello.c
```

Thread ID

Thread attributes (usually NULL)

Thread routine

Thread arguments (void *p)

```
void* thread(void* vargp) { /* thread routine */
    printf("Hello, world!\n");
    return NULL;
}
hello.c
```

Return value (void **p)