15-213

"The course that gives CMU its Zip!"

System-level programming II: Processes Feb 24, 2000

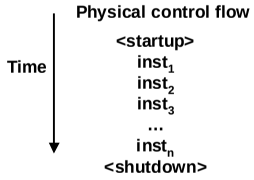
Topics

- User-level view of processes
- Exceptions
- Context switches
- Higher level control flow mechanisms

Control flow

From startup to shutdown, a CPU simply reads and executes a sequence of instructions, one at a time.

- this sequence is the system's physical *control flow* (or *flow of control*).

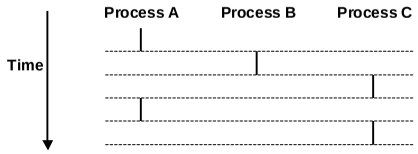


Processes

Modern operating systems partition the physical control flow into a collection of *logical control flows* called processes (*task or jobs*).

Each process is an instance of a running program.

- one process for each program that runs during the system's lifetime.



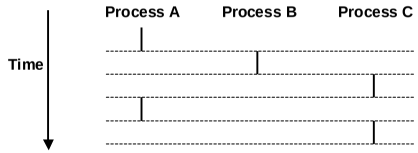
Concurrent processes

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

Otherwise, they are *sequential.*

Examples:

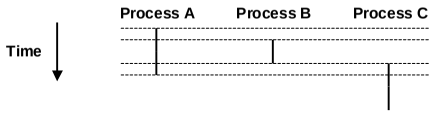
- Concurrent: A & B, A&C
- Sequential: B & C



User view of concurrent processes

Control flows for concurrent processes are physically disjoint in time.

However, we can think of concurrent processes are running in parallel with each other.

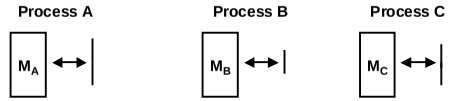


Separate address spaces

The OS also provides each process with its own address space.

- A process must take explicit steps to share part of its address space with other processes.

Provides the illusion that the process has its own memory.



Fork(): Creating new processes

`int fork(void)`

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

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


Exit(): Destroying processes

`void exit(int status)`

- exits a process
- `atexit()` registers functions to be executed upon exit

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

main() {
   atexit(cleanup);
   if (fork() == 0) {
      printf("hello from child\n");
   }
   else {
      printf("hello from parent\n");
   }
   exit();
}
```

Wait(): Synchronizing with children

`int wait(int *child_status)`

- suspends current process until one of its children terminates
- return value = 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

```
main() {
   int child_status;

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

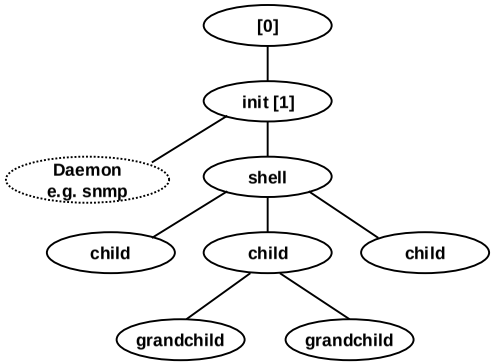

Exec(): Running new programs

`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!

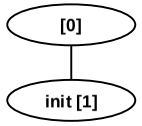
```
main() {
   if (fork() == 0) {
      execl("/usr/bin/cp","cp","foo","bar",0);
   }
   wait(NULL);
   printf("copy completed\n");
   exit();
}
```


Linux process hierarchy



Unix Startup: Step 1

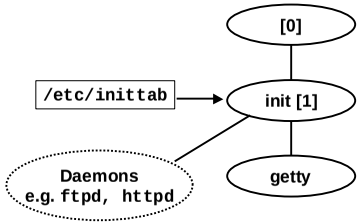
1. Pushing reset button loads the PC with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel (e.g., `/vmunix`)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.



process 0: handcrafted kernel process

process 1: user mode process
`fork()` and `exec(/sbin/init)`

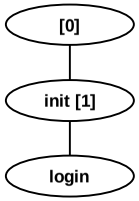
Unix Startup: Step 2



init forks new processes as per the `/etc/inittab` file

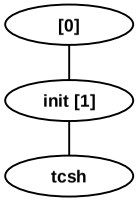
forks a getty (get tty or get terminal) for the console

Unix Startup: Step 3



getty execs a login program

Unix Startup: Step 4



login gets user's login and password
if OK, it execs a shell
if not OK, it execs another getty

Example: Loading and running programs from a shell

```
/* read command line until EOF */
while (read(stdin, buffer, numchars)) {
   <parse command line>
   if (<command line contains '&'>)
      background_process = TRUE;
   else background_process = FALSE;

   /* for commands not in the shell command language */
   if (fork() == 0) {
      execl(cmd, cmd, 0)
   }
   if (!background_process)
      retpid = wait(&status);
}
```

Example: Concurrent network server

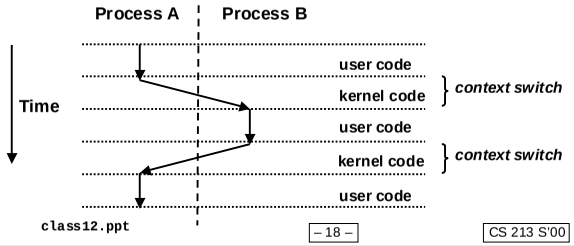
```
void main()  {
  master_sockfd = sl_passivesock(port); /* create master socket */
  while (TRUE) {
    worker_sockfd = sl_acceptsock(master_sockfd); /* await request */
    switch (fork()) {
      case 0: /* child closes its master and manipulates worker */
        close(master_sockfd);
        /* code to read and write to/from worker socket goes here */
        exit(0);

      default: /* parent closes its copy of worker and repeats */
        close(worker_sockfd);

      case -1: /* error */
        fprintf("fork error\n");
        exit(0);
    }
  }
}
```


Implementing processes

Processes are managed by a shared piece of OS code called the *kernel*

- **the kernel is not a separate process, but rather runs as part of user process**

Question: How does the control flow change from one process to another?



Altering the control flow

So far in class, we've discussed two mechanisms for changing the control flow:

- **jumps**
- **call and return using the stack discipline.**

These are insufficient for a useful system

- **only instructions can change control flow.**
- **difficult for the CPU to react to internal or external events.**
  - data arriving from a disk or a network adapter.
  - divide by zero
  - user hitting ctl-c
- **no way for the OS to preempt one process for another.**
  - processes must explicitly pass control to the OS.
    - » cooperative multitasking a la Windows 3.1.

Advanced mechanisms for altering control flow

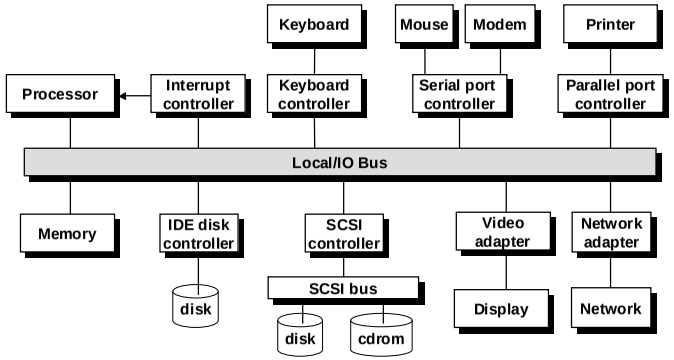
Low level mechanism:

- **exceptions**
  - change in control flow in response to an internal or external event
- **implemented as a combination of both hardware and OS software**

Higher level mechanisms:

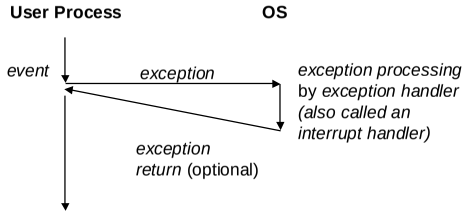
- **process context switch**
- **process receiving a signal**
- **nonlocal jumps (setjmp/longjmp)**
- **these higher level mechanisms are implemented by either:**
  - OS software (context switch and signals).
  - Language runtime library: nonlocal jumps.

System context for exceptions



Exceptions

An *exception* is a transfer of control to the OS in response to some *event* (i.e. change in processor state)



Internal (CPU) exceptions

Internal exceptions occur as a result of events generated by executing instructions.

- **Execution of a system call instruction.**
  - allows a program to ask for OS services (e.g., timer updates)
- **Execution of a break instruction**
  - used by debuggers
- **Errors during instruction execution**
  - arithmetic overflow, address error, parity error, undefined instruction
- **Events that require OS intervention**
  - virtual memory page fault

External exceptions (interrupts)

External exceptions occur as a result of events generated by devices external to the processor (managed by interrupt controller).

- **I/O interrupts**
  - hitting ctl-c at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector from a disk
- **Hard reset interrupt**
  - hitting the reset button
- **Soft reset interrupt**
  - hitting ctl-alt-delete on a PC

System calls (internal exceptions)

System calls (traps) are *expected* program events

- e.g., `fork(), exec(), wait(), getpid()`

User code:

- calls a user-level library function
- which executes a special `syscall` instruction
  - e.g., `syscall(id)`
- results in a switch from *user mode* to *kernel mode*
- transfers control to a kernel *system call interface*

System call interface:

- find entry in `syscall` table corresponding to `id`
- determine number of parameters
- copy parameters from user memory to kernel memory
- save current process context (in case of abortive return)
- invoke appropriate function in kernel

Context of a process

Context of a process:

- state that is necessary to restart the process if it is interrupted
- includes the following:
  - user-level context
  - register context
  - system-level context

User-level context:

- `text`, `data`, and `bss` segments, plus user stack
- includes arguments and environment variables

Register context:

- PC, general purpose regs and floating point regs, IEEE rounding mode, kernel stack pointer, process table address, etc.

System-level context:

- various OS tables, process and memory tables, kernel stack, etc.

Context switches

The kernel may decide to pass control to another process when the current process does one of the following:

- puts itself to sleep
  - e.g., after initiating an I/O request to a slow device
- exits
- returns from a system call
- returns after being interrupted by an I/O device

A *context switch* passes control to a new process via the following steps:

- save the current process context
- select a new process to run (scheduling)
- restore the (previously saved) context of the new process
- pass control to the new process

Higher level mechanisms for altering the control flow

Signals

- signals are software events generated by OS and processes
  - an OS abstraction for exceptions and interrupts
- signals are sent from the kernel or processes to other processes.
- different signals are identified by small integer ID's
  - e.g., SIGINT: sent to foreground process when user hits ctl-c
  - e.g., SIGALRM: sent to process when a software timer goes off
- the only information in a signal is its ID and the fact that it arrived.

Signal handlers

- programs can install signal handlers for different types of signals
  - handlers are asynchronously invoked when their signals arrive.

See text for more details

A program that reacts to externally generated events (ctrl-c)

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

static void handler(int sig) {
  printf("You think hitting ctrl-c will stop the bomb?\n");
  sleep(2);
  printf("Well...");
  fflush(stdout);
  sleep(1);
  printf("OK\n");
  exit(0);
}

main() {
  signal(SIGINT, handler); /* installs ctl-c handler */
  while(1) {
  }
}
```


A program that reacts to internally generated events

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

int beeps = 0;

/* SIGALRM handler */
void handler(int sig) {
  printf("BEEP\n");
  fflush(stdout);

  if (++beeps < 5)
    alarm(1);
  else {
    printf("BOOM!\n");
    exit(0);
  }
}
```

```
main() {
  signal(SIGALRM, handler);
  alarm(1); /* send SIGALRM in
               1 second */

  while (1) {
    /* handler returns here */
  }
}
```

```
bass> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!
bass>
```


Nonlocal jumps: setjmp()/longjmp()

Powerful (but dangerous) user-level mechanism for transferring control to an arbitrary location.

- controlled to way to break the procedure stack discipline
- useful for error recovery

`int setjmp(jmp_buf j)`

- must be called before longjmp
- identifies a return site for a subsequent longjmp.

setjmp implementation:

- remember where you are by storing the current register context, stack pointer, and PC value in jmp_buf.
- return 0

setjmp/longjmp (cont)

`void longjmp(jmp_buf j, int i)`

- meaning:
  - return from the setjmp remembered by jump buffer j again...
  - ...this time returning i
- called after setjmp
- a function that never returns!

longjmp Implementation:

- restore register context from jump buffer j
- set %eax (the return value) to i
- jump to the location indicated by the PC stored in jump buf j.

setjmp/longjmp example

```
#include <setjmp.h>
jmp_buf buf;

main() {
   if (setjmp(buf) != 0) {
      printf("back in main due to an error\n");
   else
      printf("first time through\n");
   p1(); /* p1 calls p2, which calls p3 */
}
...
p3() {
   <error checking code>
   if (error)
      longjmp(buf, 1)
}
```


Putting it all together: A program that restarts itself when ctrl-c'd

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

sigjmp_buf buf;

void handler(int sig) {
  siglongjmp(buf, 1);
}

main() {
  signal(SIGINT, handler);

  if (!sigsetjmp(buf, 1))
    printf("starting\n");
  else
    printf("restarting\n");
```

```
while(1) {
    sleep(1);
    printf("processing...\n");
  }
}
```

```
bass> a.out
starting
processing...
processing...
restarting      <----Ctrl-c
processing...
processing...
processing...
restarting      <----Ctrl-c
processing...
restarting      <----Ctrl-c
processing...
processing...
```