15-213

"The course that gives CMU its Zip!"

Exceptional Control Flow
Oct 24, 2000

Topics

- **Exceptions**
- **Process context switches**
- **Signals**
- **Non-local jumps**

class17.ppt

Control flow

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

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

Physical control flow

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

Altering the control flow

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

- **jumps and branches**
- **call and return using the stack discipline.**
- **both react to changes in program state.**

These are insufficient for a useful system

- **difficult for the CPU to react to changes in system state.**
  - data arrives from a disk or a network adapter.
  - instruction divides by zero
  - user hitting ctl-c at the keyboard
  - system timer expires

Real systems need mechanisms for "exceptional control flow"

Exceptional control flow

Mechanisms for exceptional control flow exists at all levels of a computer system.

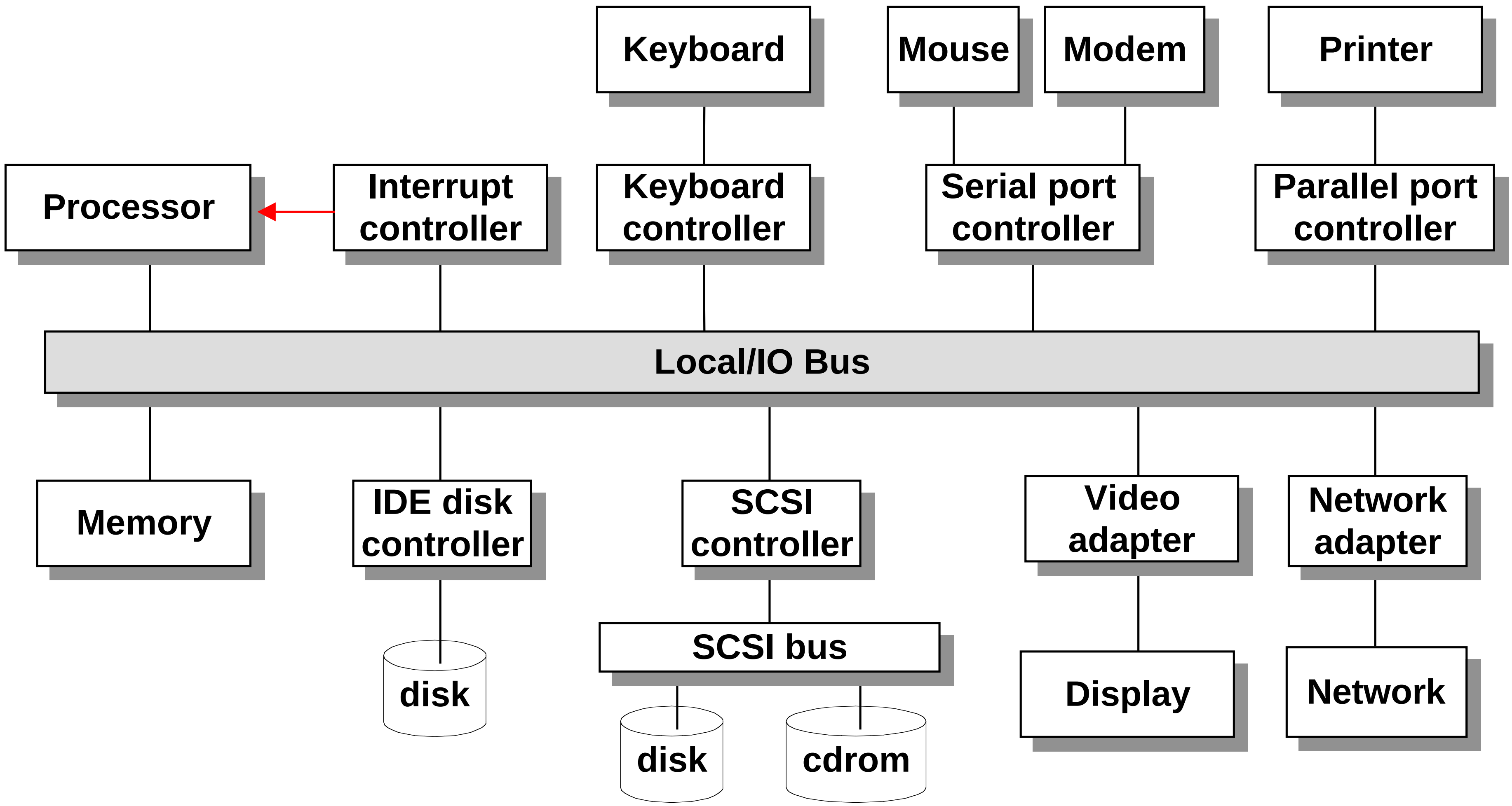
Low level mechanism:

- **exceptions**
  - change in control flow in response to a system event (i.e., change in system state)
- **implemented as a combination of both hardware and OS software**

Higher level mechanisms:

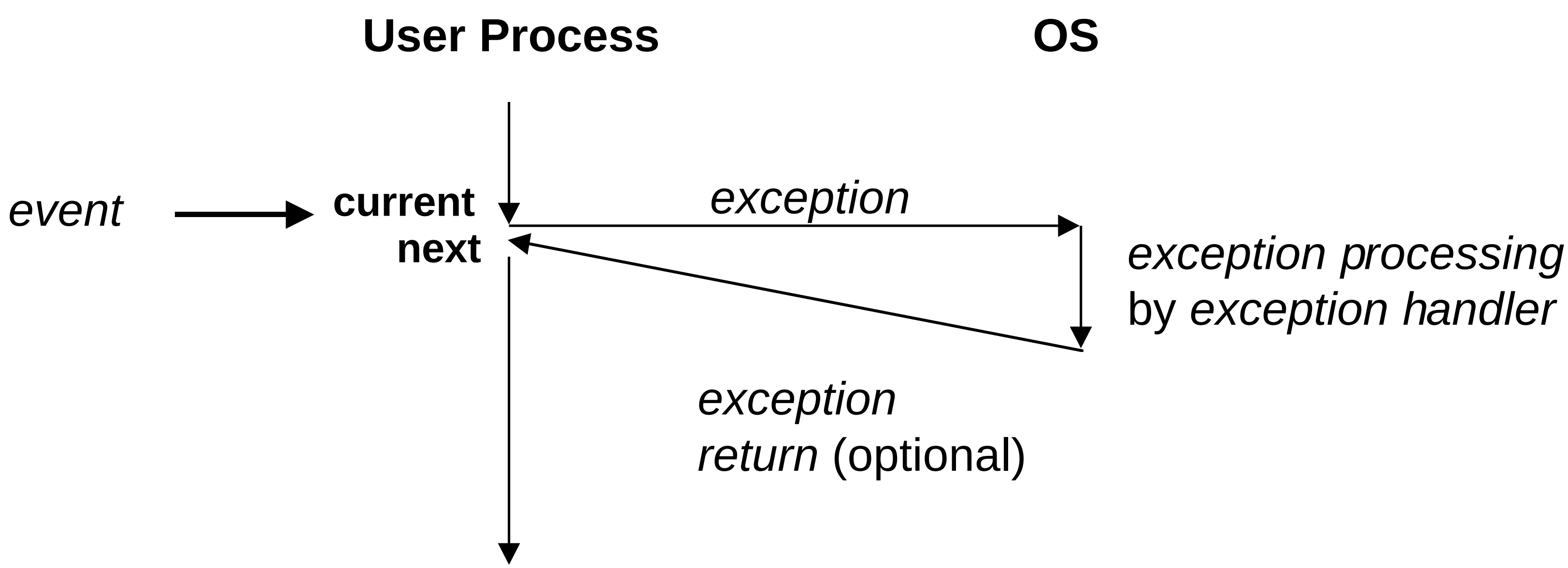
- **process context switch**
- **signals**
- **nonlocal jumps (setjmp/longjmp)**
- **implemented by either:**
  - OS software (context switch and signals).
  - C language runtime library: nonlocal jumps.

System context for exceptions

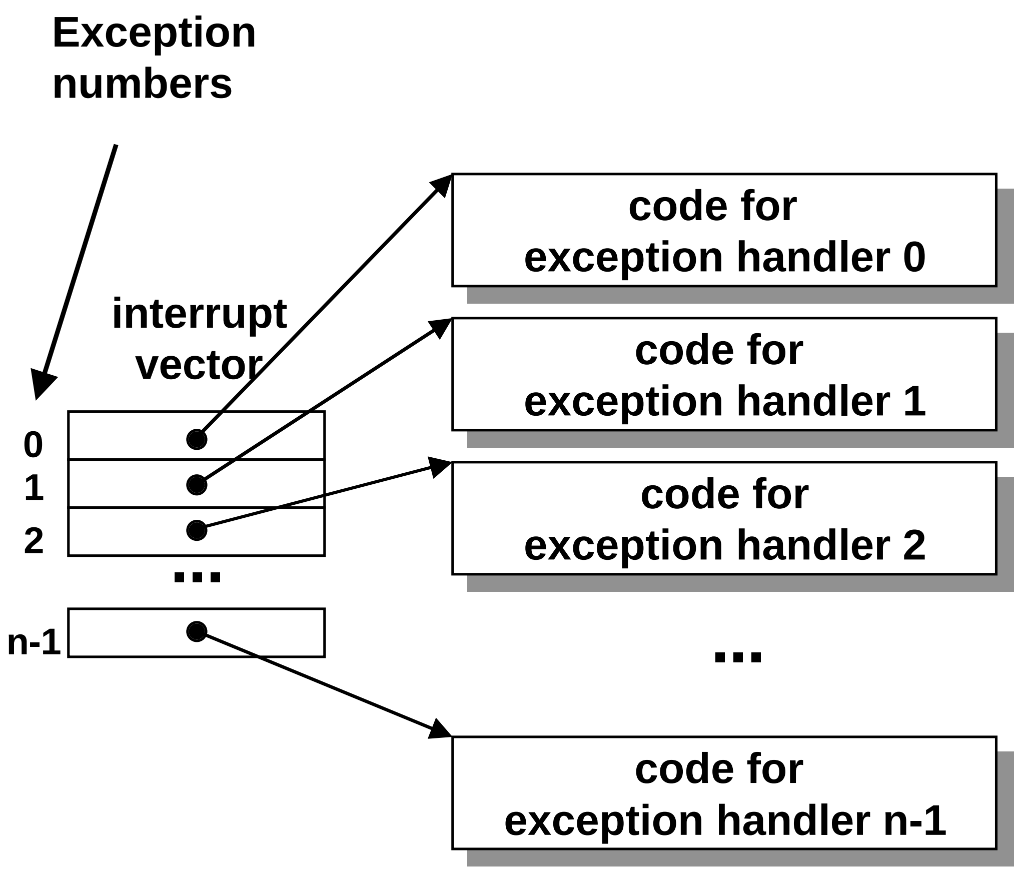


Exceptions

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



Interrupt vectors



1. Each type of event has a unique exception number k

2. Jump table (interrupt vector) entry k points to a function (exception handler).

3. Handler k is called each time exception k occurs.

Asynchronous exceptions (interrupts)

Caused by events (changes in state) external to the processor

- **Indicated by setting the processor's interrupt pin**
- **handler returns to "next" instruction.**

Examples:

- **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

Synchronous exceptions

Caused by events (changes in state) that occur as a result of executing an instruction:

- **Traps**
  - intentional
  - returns control to “next” instruction
  - Examples: system calls, breakpoint traps
- **Faults**
  - unintentional but possibly recoverable
  - either re-executes faulting (“current”) instruction or aborts.
  - Examples: page faults (recoverable), protection faults (unrecoverable).
- **Aborts**
  - unintentional and unrecoverable
  - aborts current program
  - Examples: parity error, machine check.

Processes

Def: A *process* is an instance of a running program.

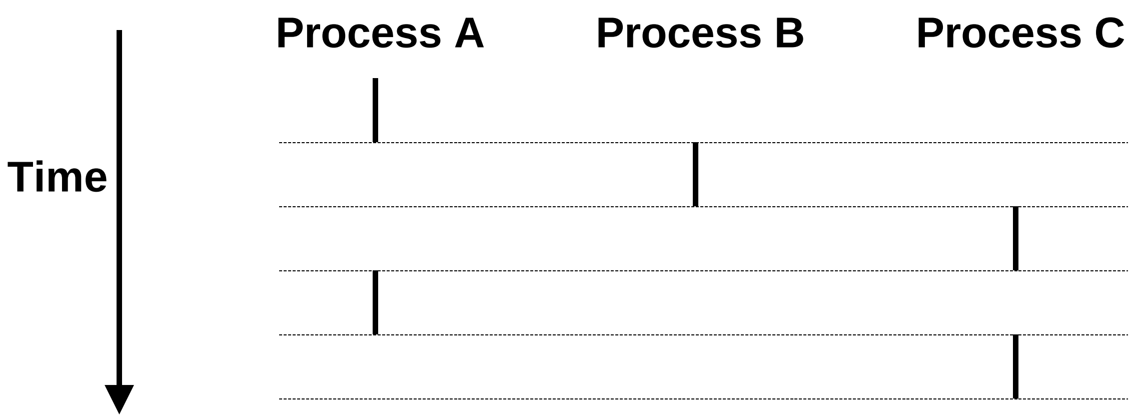
- **One of the most profound ideas in computer science.**

Process provides each program with two key abstractions:

- **Logical control flow**
  - gives each program the illusion that it has exclusive use of the CPU.
- **Private address space**
  - gives each program the illusion that has exclusive use of main memory.

Logical control flows

Each process has its own logical control flow



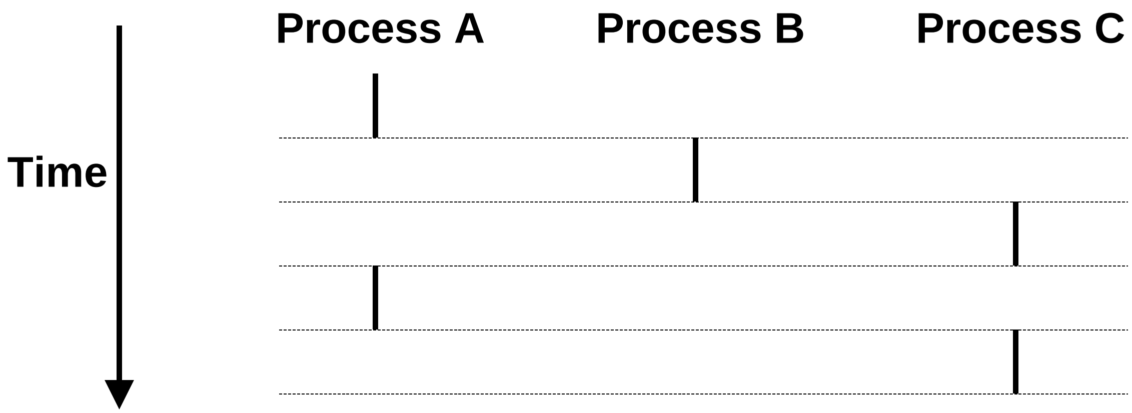
Concurrent processes

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

Otherwise, they are *sequential.*

Examples:

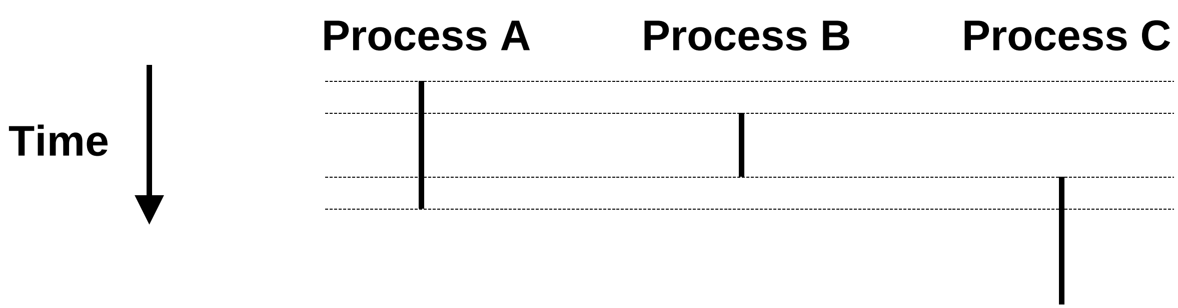
- **Concurrent: A & B, A&C**
- **Sequential: B & C**



User view of concurrent processes

Control flows for concurrent processes are physically disjoint in time.

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

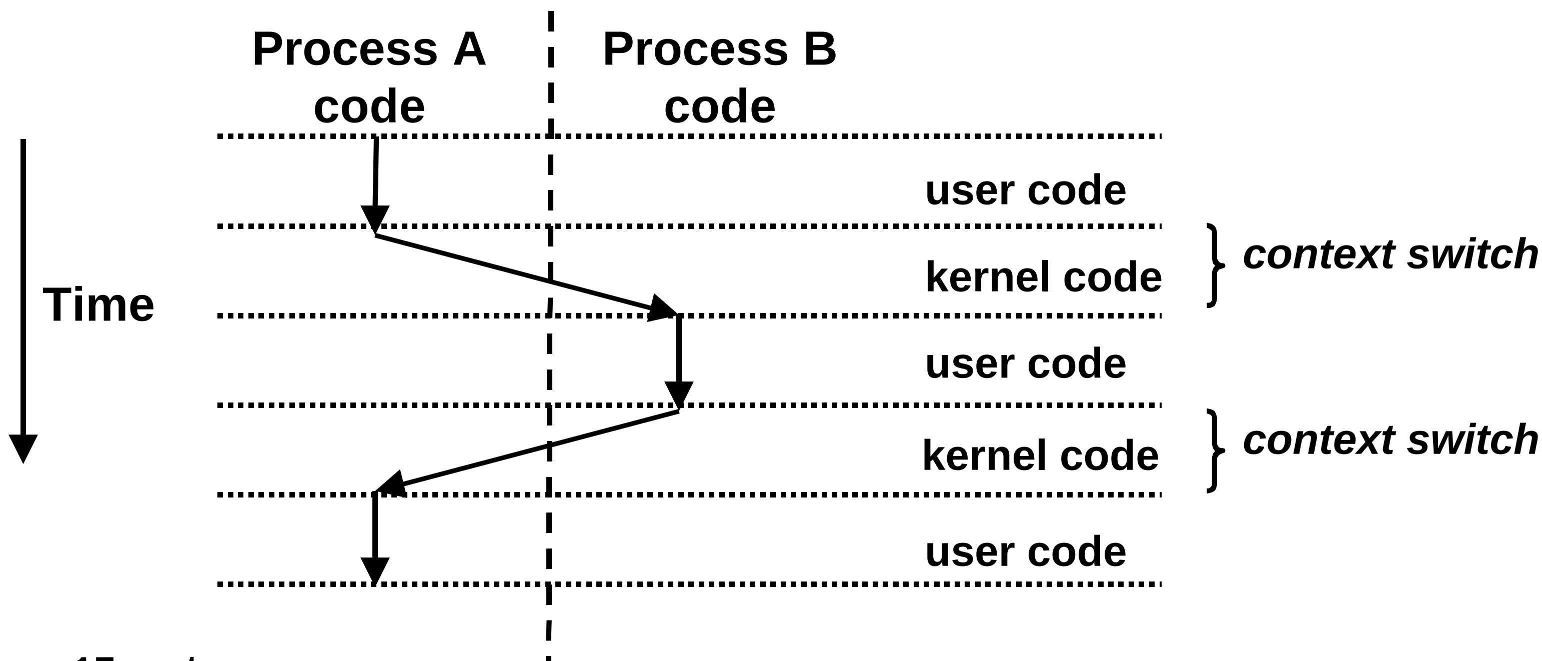


Context switching

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

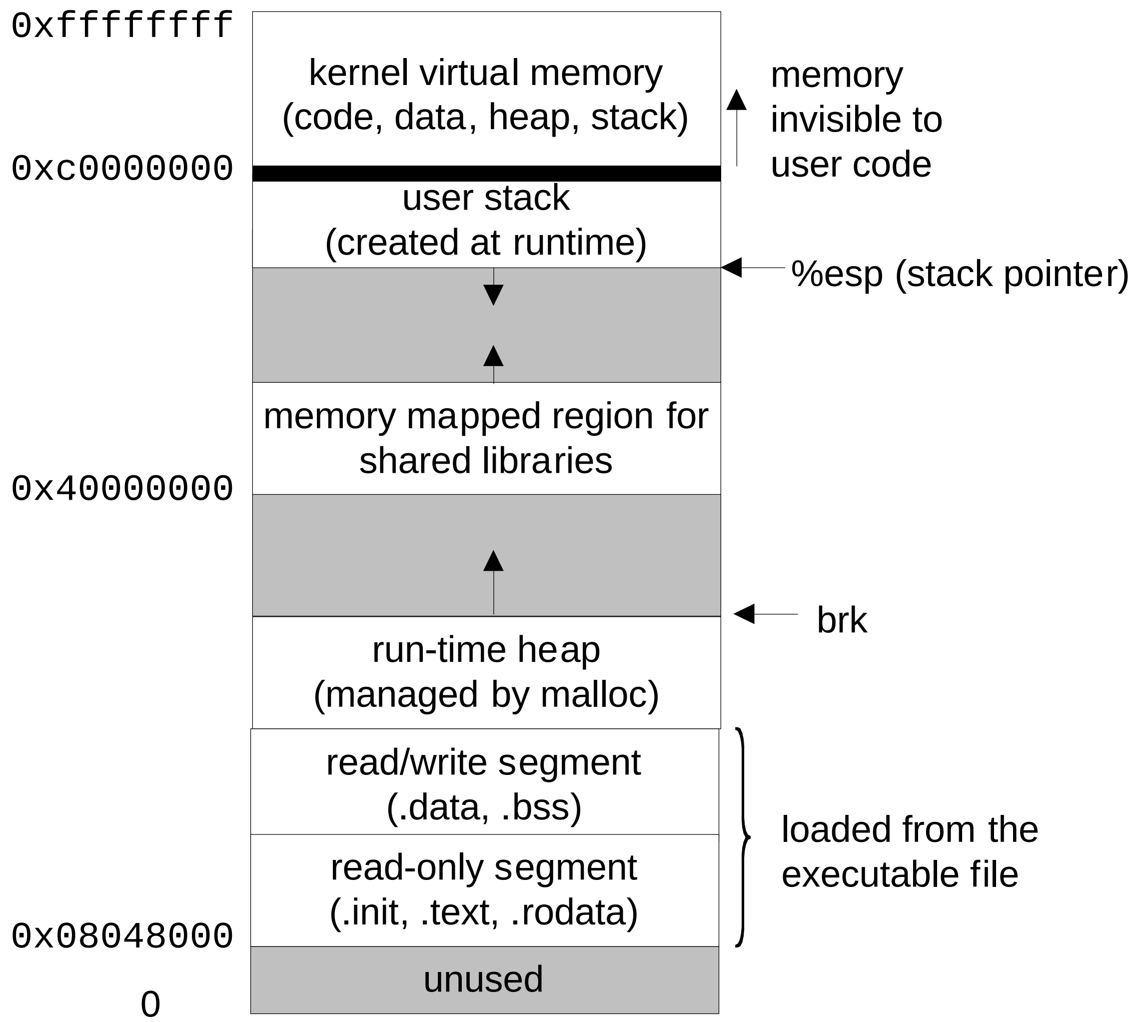
- **Important: the kernel is not a separate process, but rather runs as part of some user process**

Control flow passes from one process to another via a *context switch.*



Private address spaces

Each process has its own private address space.



`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");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

`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();
}
```

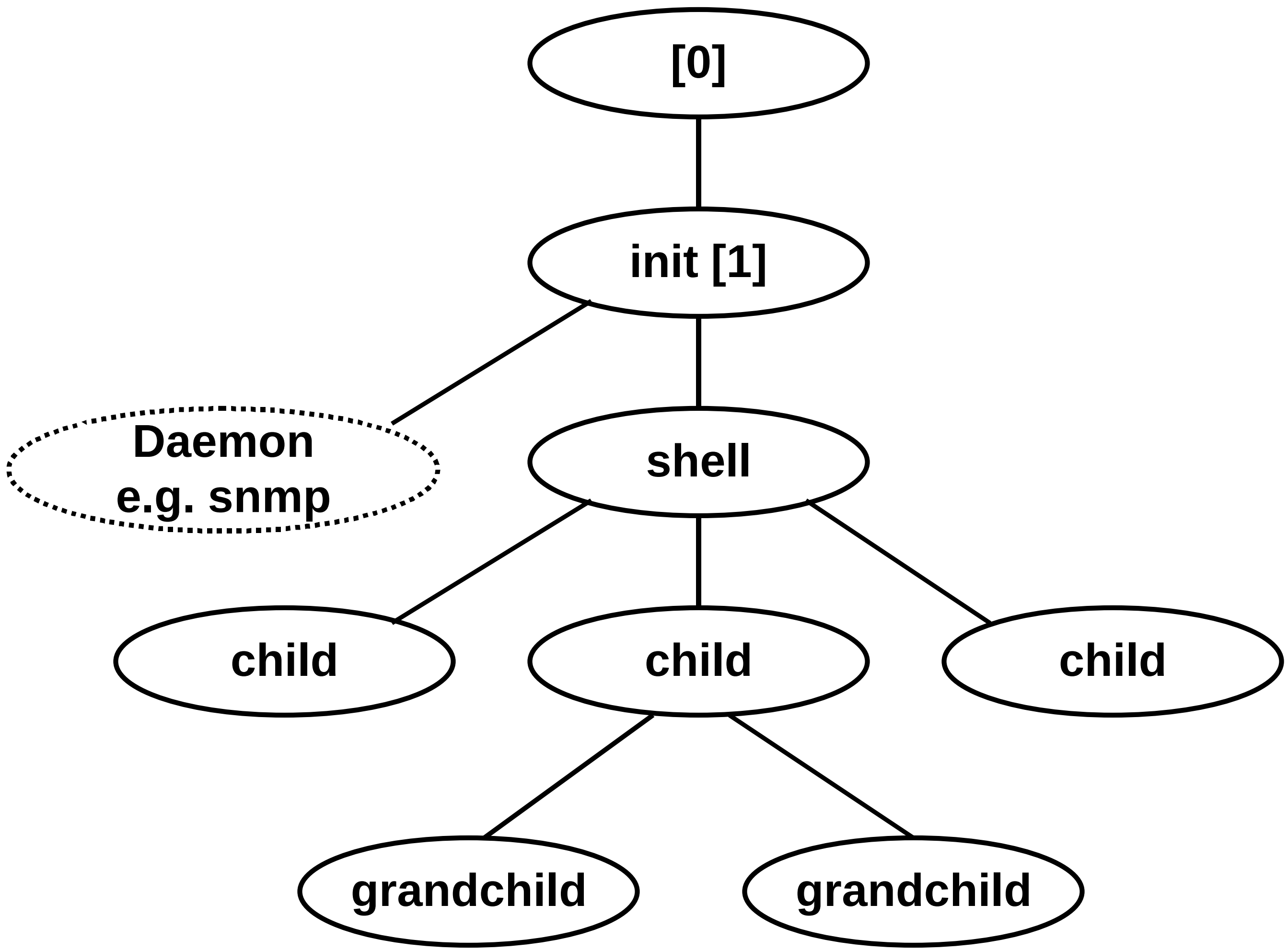
exec: Running new programs

`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!**

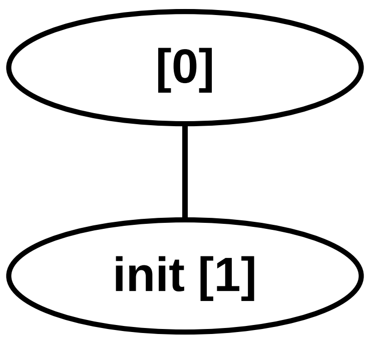
```
main() {
   if (fork() == 0) {
      execl(“/usr/bin/cp”,“cp”,“foo”,“bar”,0);
   }
   wait(NULL);
   printf(“copy completed\n”);
   exit();
}
```

Linux process hierarchy



Linux Startup: Step 1

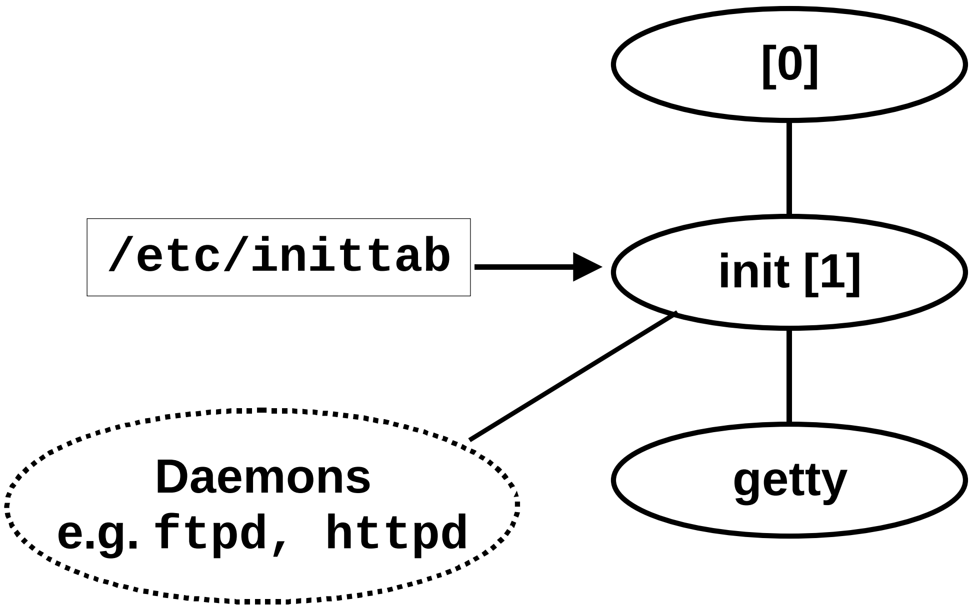
1. Pushing reset button loads the `PC` with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel (e.g., `/vmunix`)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.



process 0: handcrafted kernel process

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

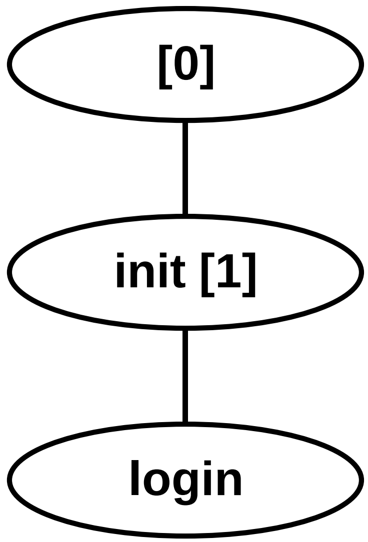
Linux Startup: Step 2



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

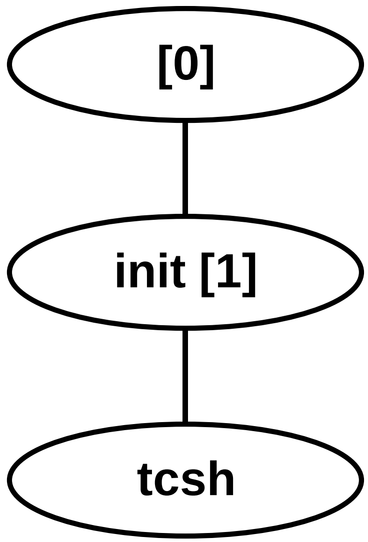
forks a getty (get tty or get terminal) for the console

Linux Startup: Step 3



getty execs a login program

Linux 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);
    }
  }
}
```

Signals

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 handout 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...
```