15-213

Introduction to Computer Systems

Program Translation and Execution II: Processes
Oct 1, 1998

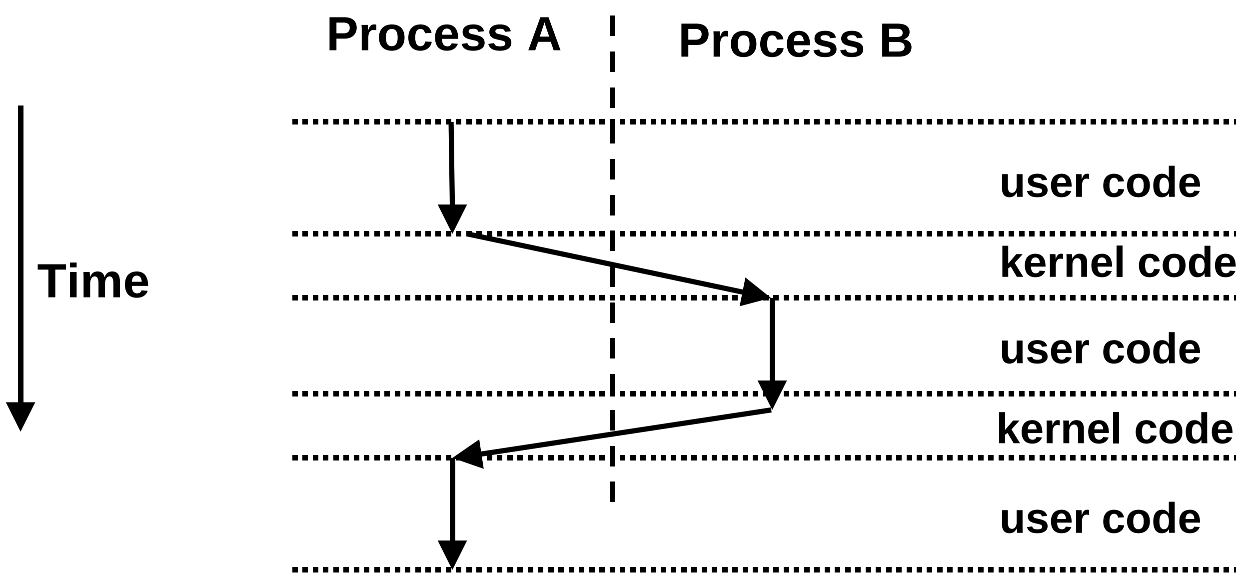
Topics

- User-level view of processes
- Implementation of processes
- setjmp/longjmp

Processes

A process is an instance of a running program

- **runs concurrently with other processes (*multitasking*)**
- **managed by a shared piece of OS code called the *kernel***
  - kernel is not a separate process, but rather runs as part of some user process
- **each process has its own data space and process id (pid)**
- **data for each protected protected from other processes**



Just a stream of instructions!

Fork

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

void exit(int status)

- **exits a process**
- **atexit() function registers functions to be executed on 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

int wait(int child_status)

- **waits for a child to terminate and returns status and pid**

```
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

int execl(char *path, char *arg0, char *arg1, ...)

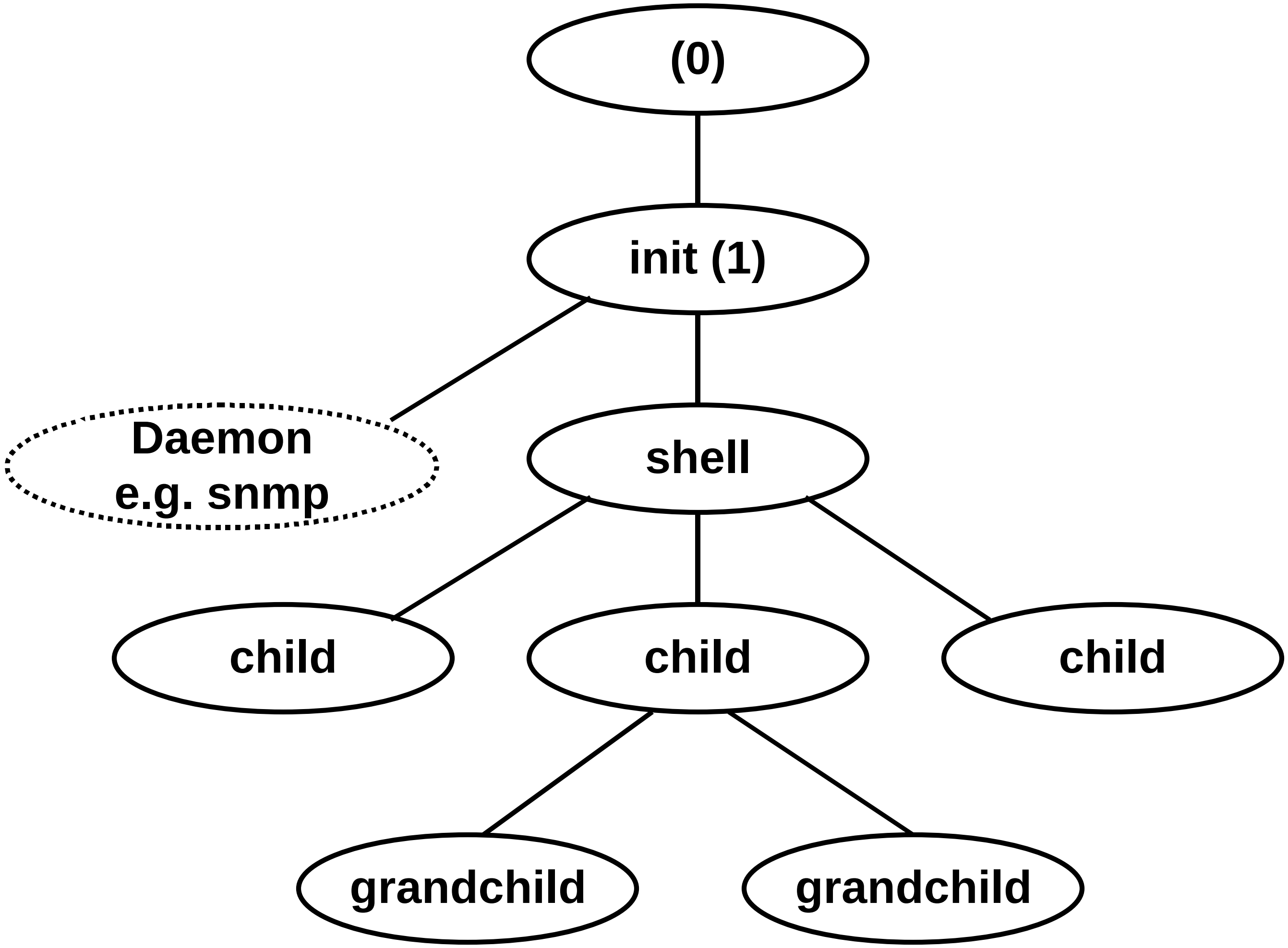
- loads and runs executable at path with args arg0, arg1, ...
- returns -1 if error, otherwise doesn’t return!

```
main() {

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

Example: Concurrent network server

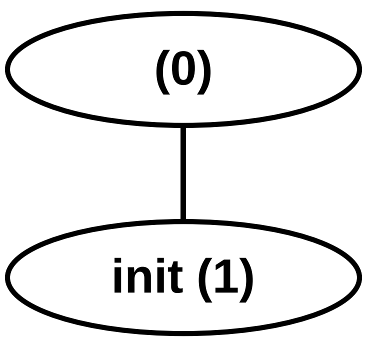
```
void main()  {
  master_sockfd = sl_passivesock(port); /* create master socket */
  while (1) {
    slave_sockfd = sl_acceptsock(master_sockfd); /* await request */
    switch (fork()) {

      case 0: /* child closes its master and manipulates slave */
        close(master_sockfd);
        /* code to read and write to/from slave socket goes here */
        exit(0);

      default: /* parent closes its copy of slave and repeats */
        close(slave_sockfd);

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

Process hierarchy



Unix startup (1)

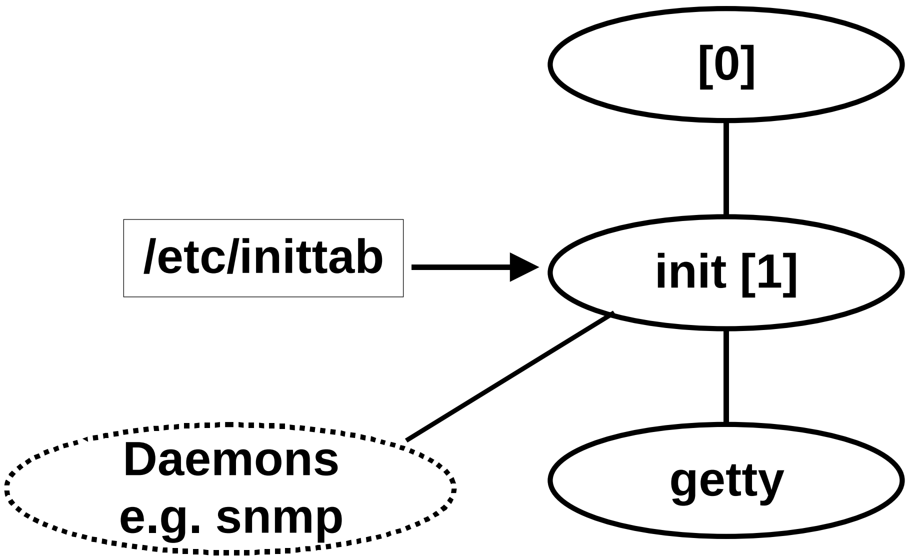
1. Pushing reset button loads the pc with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel (e.g., /vmunix)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.



process 0: handcrafted kernel process

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

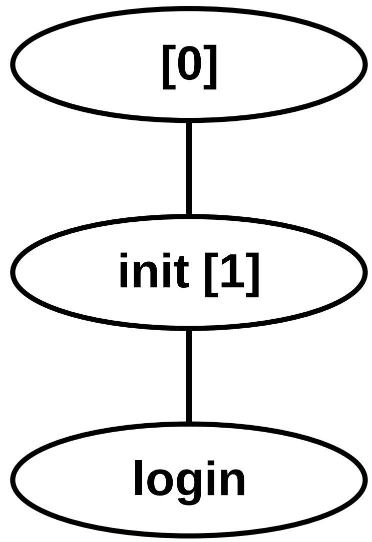
Unix startup (2)



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

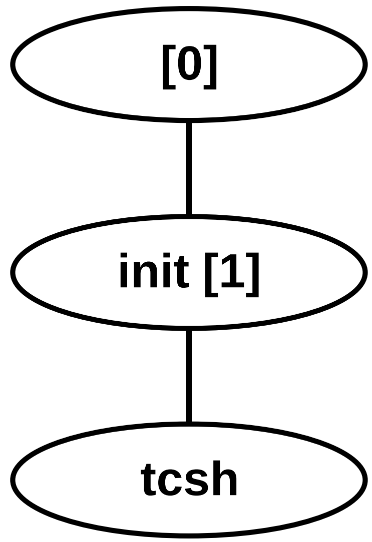
forks a getty (get tty or get terminal) for the console

Unix startup (3)



getty execs a login program

Unix startup (4)

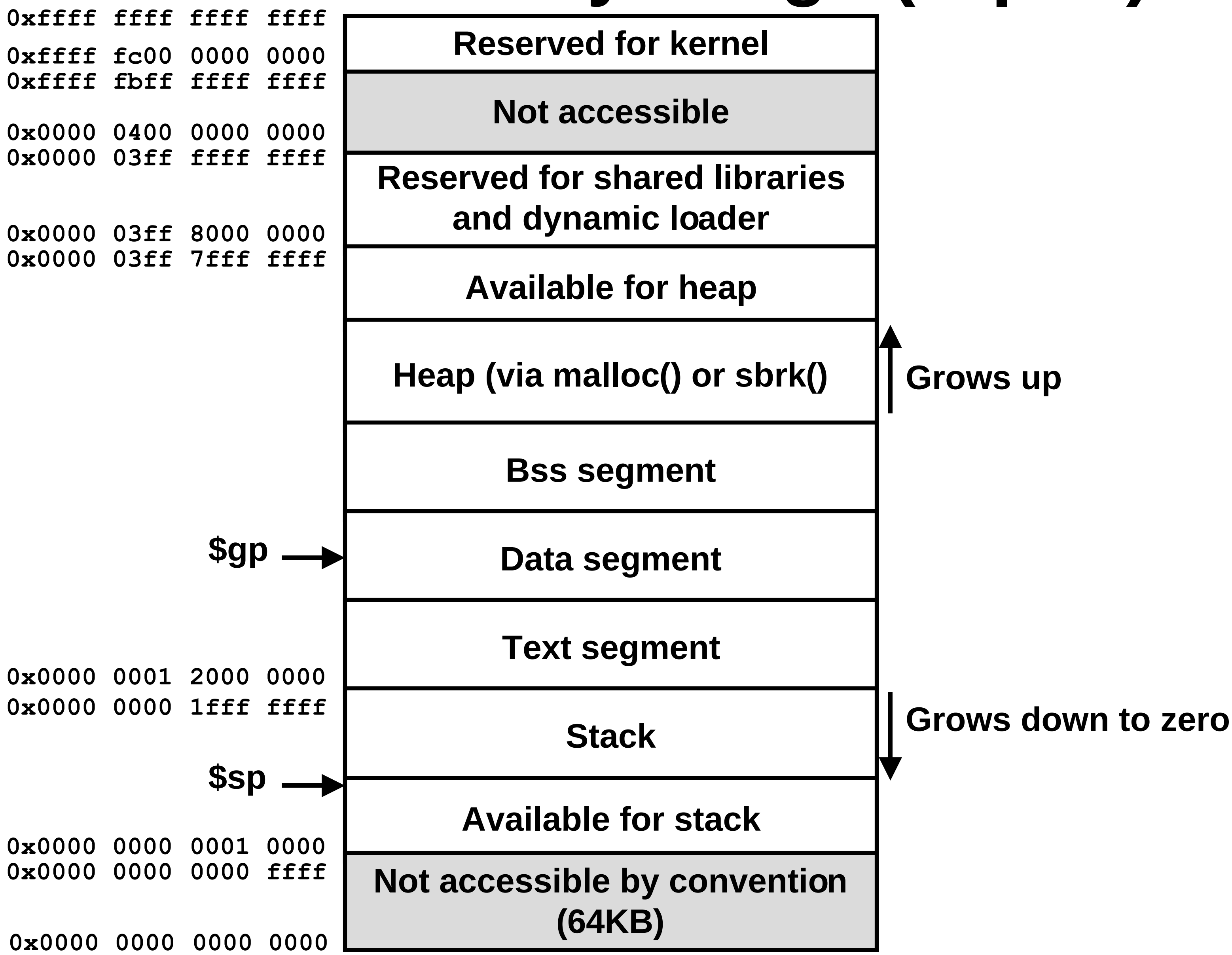


login gets user's login and passw
if OK, it execs a shell
if not OK, it execs another getty

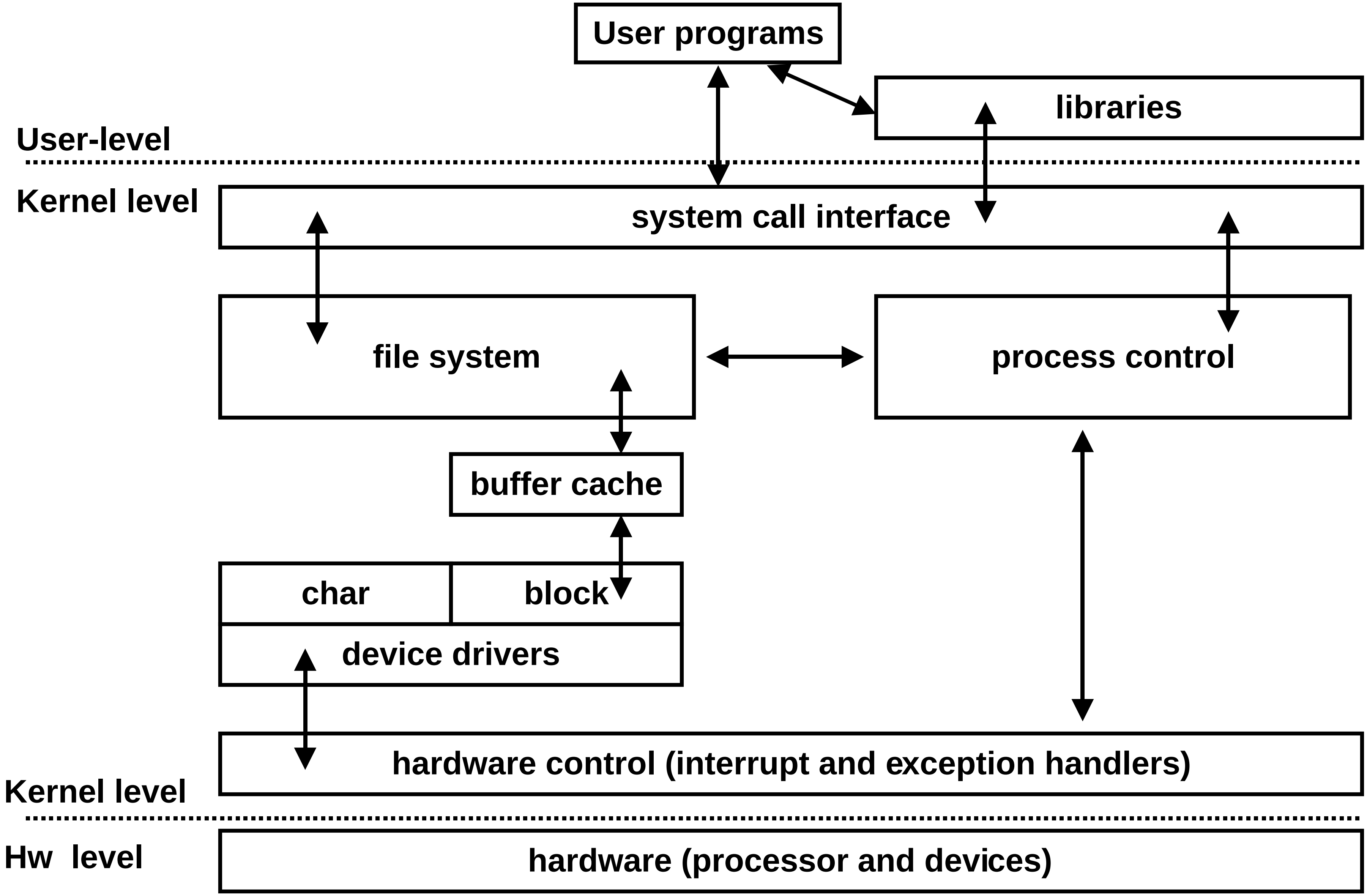
Loading and running programs from a shell

```
/* read command line until EOF */
while (read(stdin, buffer, numchars)) {
   <parse command line>
   if (<command line contains ‘&’ >)
      amper = 1;
   else
      amper = 0;
   }

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

Process memory image (Alpha)



Kernel block diagram



User and kernel modes

User mode

- **Process can**
  - execute its own instructions and access its own data.
- **Process cannot**
  - execute kernel instructions or privileged instructions (e.g. halt)
  - access kernel data or data from other processes.

Kernel mode

- **Process can**
  - execute kernel instructions and privileged instructions
  - access kernel and user addresses

Processes transition from user to kernel mode via

- ***interrupts* and *exceptions***
- ***system calls (traps)***

System call interface

System calls (traps) are *expected* program events

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

User code

- call user-level library function,
- executes special syscall instruction
  - e.g. syscall(id)
- switch from user mode to kernel mode
- transfer control to kernel system call interface

System call interface

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

Hardware control

Interrupts and exceptions are *unexpected* hardware *events*

Interrupts

- **events external to the processor**
  - I/O device asking for attention
  - timer interrupt
- **typically indicated by setting an external pin**

Exceptions

- **events internal to processor (as a result of executing an instruction)**
  - divide by zero

Same mechanism handles both

- **Interrupt or exception triggers transfer of control from user code to interrupt handlers in the hardware control part of the kernel**
- **kernel services interrupt or exception**
- **If a timer interrupt, kernel might decide to give control to a new process (context switch)**

Process control: Context of a process

The context of a process is the state that is necessary to restart the process if its interrupted. Union of ...

- user-level context
- register context
- system-level context.

User-level context

- text, data, and bss segments, and user stack

Register context

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

System-level context

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

Process control: Context switch

The kernel can decide to pass control to another process if:

- **the current process puts itself to sleep**
- **the current process exits**
- **when the current process returns from a system call**
- **when the current process returns after being interrupted**

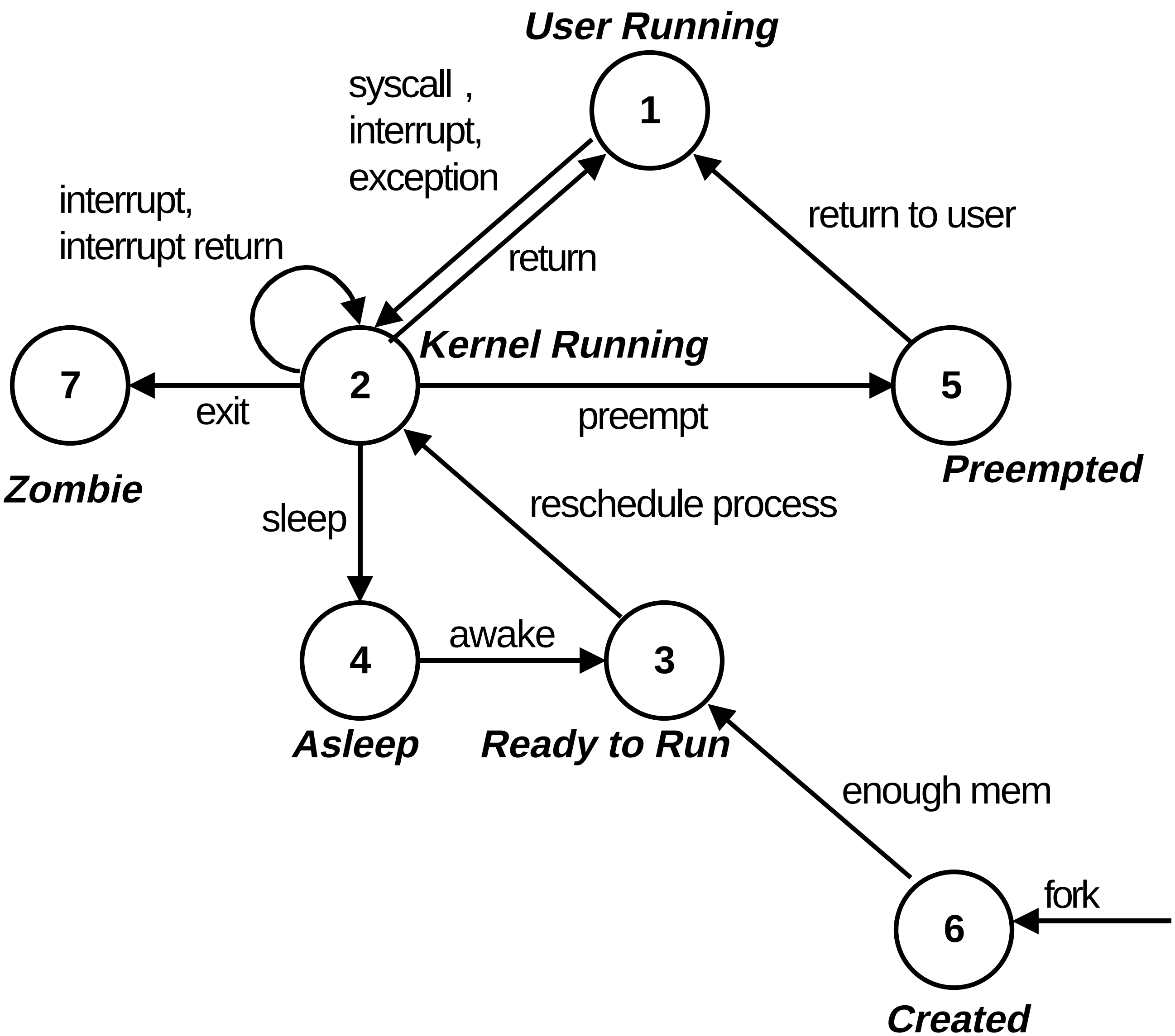
Control passed to new process via *context switch:*

- **save current process context.**
- **select new process (scheduling)**
- **restore (previously save) context of new process**
- **pass control to new process**

Process control: Process states

1. ***User Running:*** **Process is executing in user mode.**
2. ***Kernel Running:*** **Process is executing in kernel mode.**
3. ***Ready to Run:*** **Process is not executing, but is ready to as soon as the kernel schedules it.**
4. ***Asleep:*** **Process is sleeping.**
5. ***Preempted:*** **Process is returning from kernel mode to user mode, but the kernel preempts it and does a context switch to schedule another process.**
6. ***Created:*** **Process is newly created, but it is not yet ready to run, nor is it sleeping (This is the start state for all process created with fork).**
7. ***Zombie*****: The process executed the exit system call and is in the *zombie* state (until wait'ed for by its parent)**

Process states and state transitions



Setjmp/Longjmp

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

int setjmp(jmp_buf j)

- **must be called before longjmp**
- **meaning:**
  - remember where you are by storing the current register context and PC value in jmp_buf
  - return 0

void longjmp(jmp_buf j, int i)

- **called after setjmp**
- **meaning:**
  - return from the setjmp remembered by jump buffer j with a value of i
  - restores register context from jump buf j, sets register $ra to i, sets PC to the PC stored in jump buf j.

Setjmp/Longjmp example

Useful for :

- **error recovery**
- **implementing user-level threads packages**

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

main() {
   if (setjmp(buf)) {
      printf("back in main\n");
   else
      printf("first time through\n");
   p1(); /* p1->p2->p3 */
}
...
p3() {
   <error checking code>
   if (error)
      longjmp(buf, 1)
}
```