3: Processes

Last Modified:
9/12/2002 1:14:25 AM

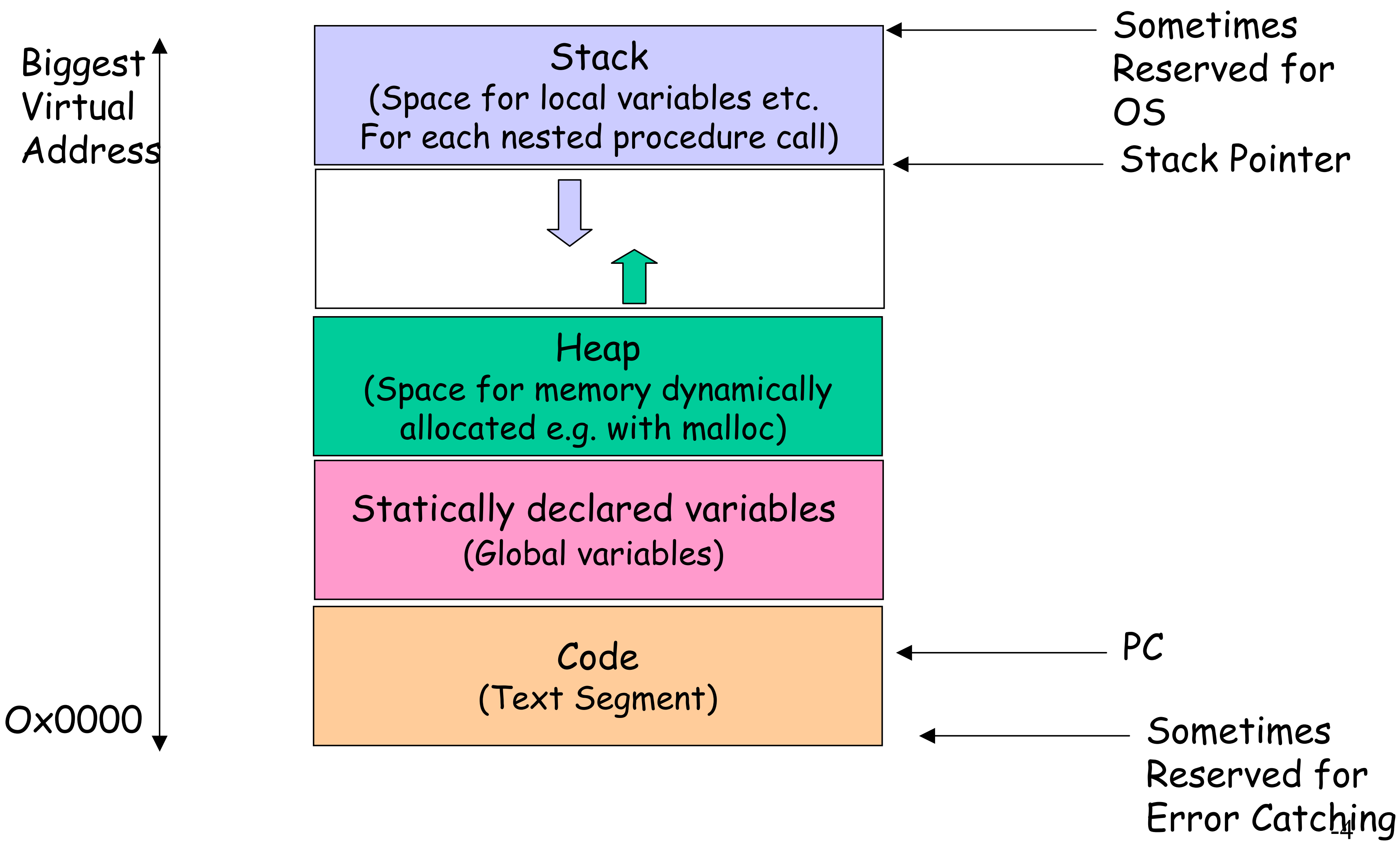
Programs vs Processes

- A program is passive
  - Sequence of commands waiting to be run
- A process is active
  - An instance of program being executed
  - There may be many processes running the same program
  - Also called job or task

What makes up a process?

- Address space
- Code
- Data
- Stack (nesting of procedure calls made)
- Register values (including the PC)
- Resources allocated to the process
  - Memory, open files, network connections

Address Space Map



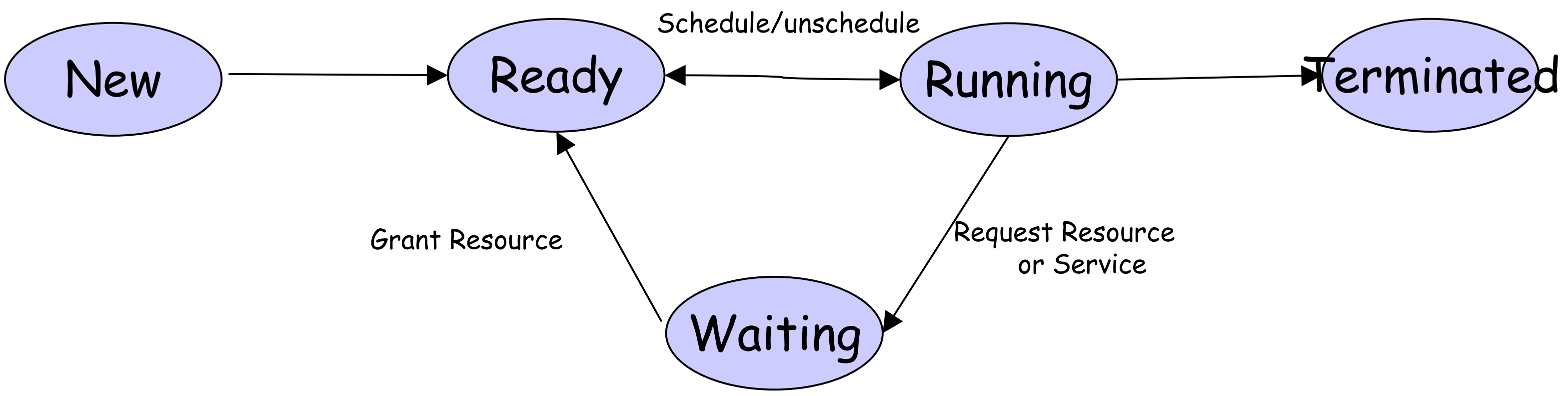
What kinds of processes are there?

- Compute bound/ IO bound
- Long-running/short-running
- Interactive/batch
- Large/small memory footprint
- Cooperating with other processes?
- ...
- How does the OS categorize processes?

Process States

- During their lifetime, processes move between various states
  - Ready – waiting for a turn to use the CPU
  - Running – currently executing on the CPU
    - How many processes can be in this state? ☺
  - Waiting – Unable to use the CPU because blocked waiting for an event
  - Terminated/Zombie – Finished executing but state maintained until parent process retrieves state

State Transitions



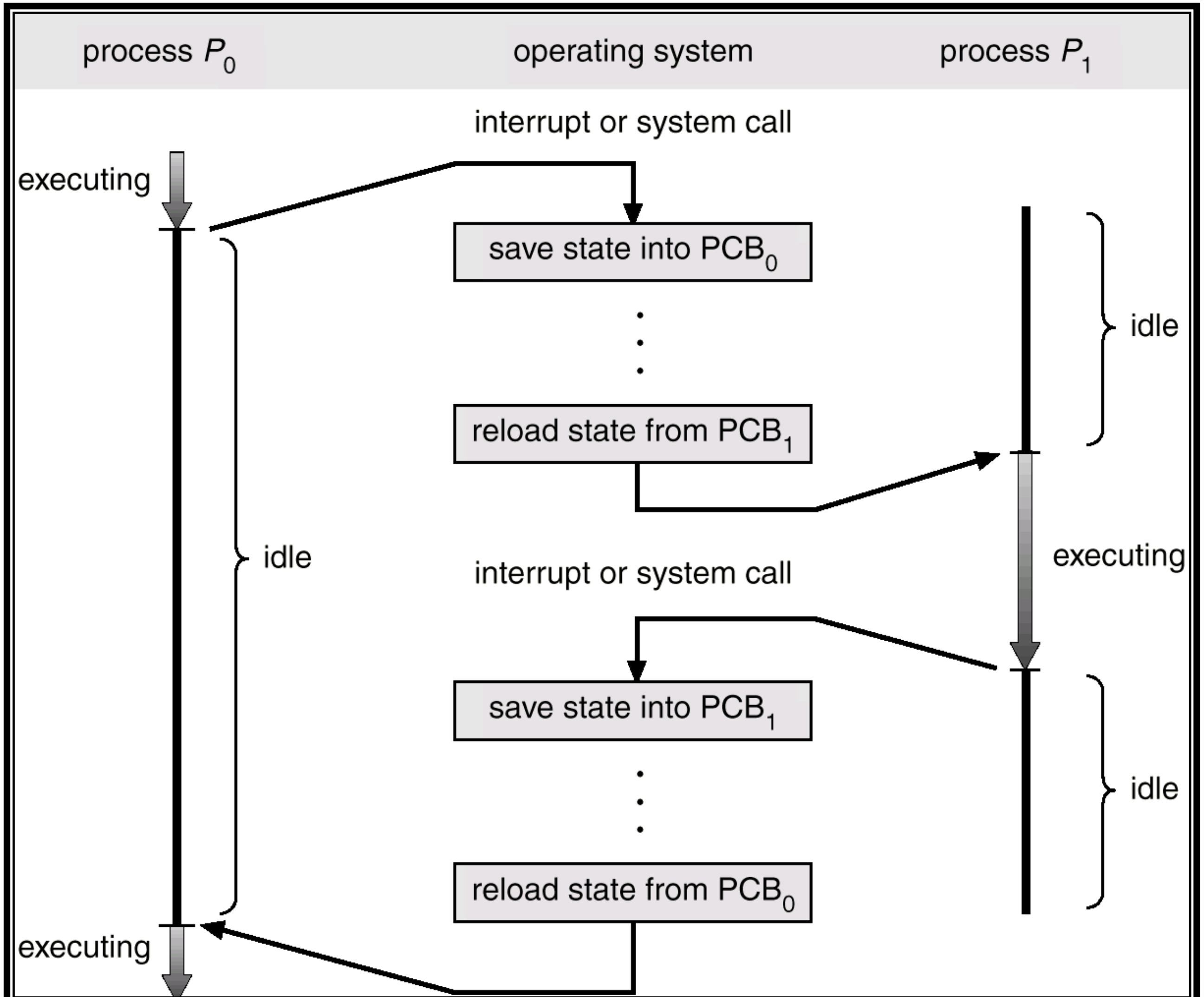
State Queues

- OSes often maintain a number of queues of processes that represent the state of the processes
  - All the runnable processes are linked together into one queue
  - All the processes blocked (or perhaps blocked for a particular class of event) are linked together
  - As a process changes state, it is unlinked from one queue and linked into another

Context Switch

- When a process is running, some of its state is stored directly in the CPU (register values, etc.)
- When the OS stops a process, it must save all of this hardware state somewhere (PCB) so that it can be restored again
- The act of saving one processes hardware state and restoring another's is called a context switch
  - 100s or 1000s per second!

Context Switch



Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue.
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU.

Schedulers (cont)

- Short-term scheduler is invoked very frequently (milliseconds) $\Rightarrow$ (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ (may be slow).
- The long-term scheduler controls the *degree of multiprogramming.*
- Processes can be described as either:
  - I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts.
  - *CPU-bound process* – spends more time doing computations; few very long CPU bursts.

Family Tree

- Age old questions – where do new processes come from?
- New processes are created when an existing process requests it
  - Creating process called the parent; created called the child
  - Children of same parent called siblings
- Children often inherit privileges/attributes from their parent
  - Working directory, Clone of address space
- When child is created, parent may either wait for it or continue in parallel

pstree

```
init-+-18*[Xvnc]
     |-amd
     |-atd
     |-bdflush
     |-crond
     |-16*[deskguide_apple]
     |-8*[gconfd-1]
     |-gedit
     |-18*[gnome-name-serv]
     |-16*[gnome-session]
     |-16*[gnome-smproxy]
     |-gnome-terminal-+-csh---gtop
     |                `-gnome-pty-helpe
     |-gnome-terminal-+-csh-+-gtop
     |                |     `-tcsh
     |                `-gnome-pty-helpe
     |-gnome-terminal-+-csh---tcsh---
       xterm---csh
     |                `-gnome-pty-helpe
     |-gpm
     |-8*[hyperbola]
     |-keventd
     |-khubd
     |-5*[kjournald]
     |-klogd
     |-ksoftirqd_CPU0
     |-ksoftirqd_CPU1
     |-kswapd
     |-kupdated
     |-lockd
```

```
init-+-18*[Xvnc]
     |-16*[magicdev]
     |-mdrecoveryd
     |-migration_CPU0
     |-migration_CPU1
     |-6*[mingetty]
     |-2*[nautilus---nautilus---8*[nautilus]]
     |-2*[nautilus---nautilus---10*[nautilus]]
     |-3*[nautilus---nautilus---9*[nautilus]]
     |-nautilus---nautilus---7*[nautilus]
     |-7*[nautilus-histor]
     |-nautilus-mozill---nautilus-mozill---4*[nautilus-
       mozill]
     |-8*[nautilus-news]
     |-8*[nautilus-notes]
     |-7*[nautilus-throbb]
     |-ntpd
     |-13*[oafd]
     |-16*[panel]
     |-portmap
     |-16*[rhn-applet]
     |-rhn-applet---gnome_segv
     |-rhnsd
     |-rpc.statd
     |-rpciod
     |-14*[sawfish]
     |-2*[sawfish---rep]
     |-scsi_eh_0
     |-scsi_eh_1
     |-sendmail
```

```
init-+-18*[Xvnc]
     |-sshd-+-2*[sshd---csh---mc]
     |      |-sshd---csh
     |      |-sshd---csh-+-more
     |      |            `-netstat
     |      `-sshd---csh---pstree
     |-syslogd
     |-16*[tasklist_applet]
     |-xemacs
     |-xfs---xfs
     |-xinetd---fam
     |-xscreensaver---greynetic
     |-xscreensaver---hopalong
     |-2*[xscreensaver---xscreensaver]
     |-xscreensaver---kumppa
     |-xscreensaver---spotlight
     |-xscreensaver---spiral
     |-xscreensaver---nerverot
     |-xscreensaver---strange
     |-xscreensaver---flame
     |-xscreensaver---grav
     |-xscreensaver---lightning
     |-xscreensaver---penetrate
     |-xscreensaver---rotzoomer---xscreensaver-ge
     |-xscreensaver---deluxe
     |-xscreensaver---rd-bomb
     |-xscreensaver---sonar
     |-xscreensaver---moire2
     `-ypbind---ypbind---2*[ypbind
```

Init process

- In last stage of boot process, kernel creates a user level process, init
- Init is the parent (or grandparent…) of all other processes
- Init does various important housecleaning activities
  - checks and mounts the filesystems, sets hostname, timezones, etc.
- Init reads various "resource configuration files" (/etc/rc.conf, etc) and spawns off processes to provide various services
- In multi-user mode, init maintains processes for each terminal port (tty)
  - Usually runs getty which executes the login program

How is a process represented?

- Usually a process or task object
- Process Control Block
- When not running how does the OS remember everything needed to start this job running again
  - Registers, Statistics, Working directory, Open files, User who owns process, Timers, Parent Process and sibling process ids
- In Linux, task_struct defined in include/linux/sched.h

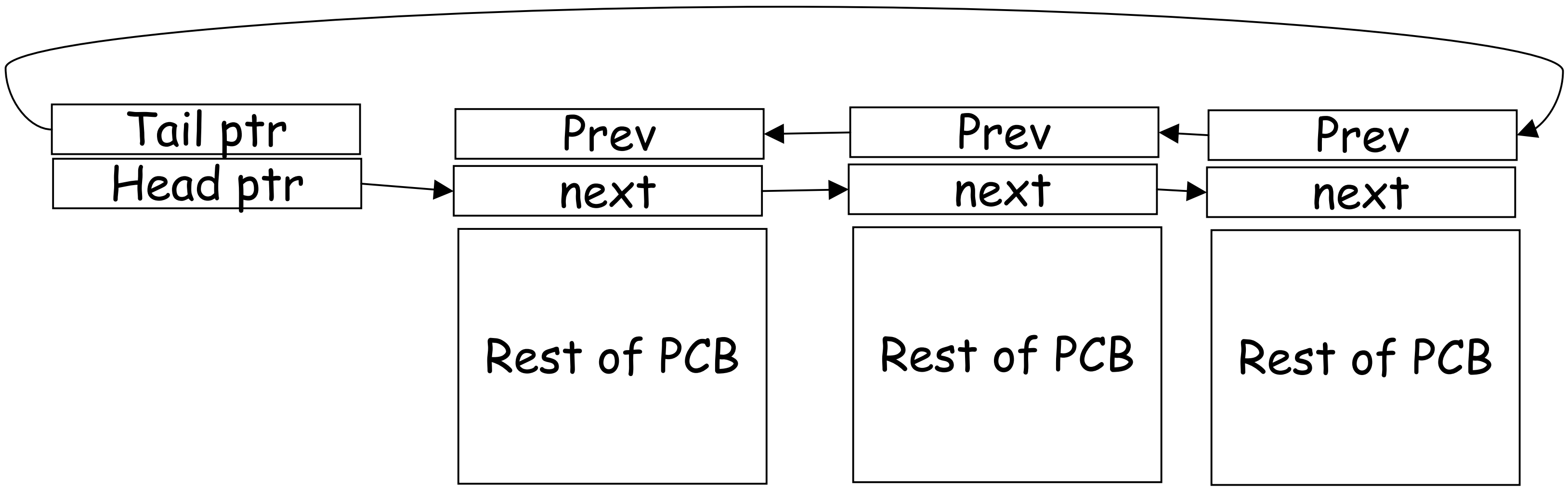
```
struct task_struct {
        /* these are hardcoded - don't touch */
        volatile long state; /* -1 unrunnable, 0 runnable, >0
stopped */
        long counter;
        long priority;
        unsigned long signal;
        unsigned long blocked; /* bitmap of masked signals
*/
        unsigned long flags; /* per process flags, defined
below */
        int errno;
        long debugreg[8]; /* Hardware debugging registers */
        struct exec_domain *exec_domain; /* various fields
*/
        struct linux_binfmt *binfmt;
        struct task_struct *next_task, *prev_task;
        struct task_struct *next_run, *prev_run;
        unsigned long saved_kernel_stack;
        unsigned long kernel_stack_page;
        int exit_code, exit_signal; /* ??? */
        unsigned long personality;
        int dumpable:1;
        int did_exec:1; /* shouldn't this be pid_t? */
        int pid;
        int pgrp; int tty_old_pgrp;
        int session; /* boolean value for session group
leader */
        int leader; int groups[NGROUPS];
        /* * pointers to (original) parent process, youngest
child, younger sibling, * older sibling,
        respectively. (p->father can be replaced with * p-
>p_pptr->pid) */
        struct task_struct *p_opptr, *p_pptr, *p_cptr,
*p_ysptr, *p_osptr;
        struct wait_queue *wait_chldexit; /* for wait4() */
        unsigned short uid,euid,suid,fsuid;
        unsigned short gid,egid,sgid,fsgid;
        unsigned long timeout, policy, rt_priority;
        unsigned long it_real_value, it_prof_value,
it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
        struct timer_list real_timer;
        long utime, stime, cutime, cstime, start_time;
        /* mm fault and swap info: this can arguably be seen
as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt, nswap, cmin_flt,
cmaj_flt, cnswap;
        int swappable:1;
        unsigned long swap_address;
        unsigned long old_maj_flt; /* old value of maj_flt */
        unsigned long dec_flt; /* page fault count of the last
time */
        unsigned long swap_cnt; /* number of pages to
swap on next pass */

        /* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];
        /* file system info */
        int link_count;
        struct tty_struct *tty; /* NULL if no tty */
        /* ipc stuff */
        struct sem_undo *semundo; struct sem_queue
*semsleeping;
        /* ldt for this task - used by Wine. If NULL,
default_ldt is used */
        struct desc_struct *ldt;
        /* tss for this task */
        struct thread_struct tss;
        /* filesystem information */
        struct fs_struct *fs;
        /* open file information */
        struct files_struct *files;
        /* memory management info */
        struct mm_struct *mm;
        /* signal handlers */
        struct signal_struct *sig;
        #ifdef __SMP__
                int processor;
                int last_processor;
                int lock_depth;   /* Lock depth. We
can context switch in and out of holding a syscall kernel lock... */
        #endif };
```

Management of PCBs

- PCBs are data structures (just like you are used to at user level)
- Space for them may be dynamically allocated as needed or perhaps a fixed sized array of PCBs for the maximum number of possible processes is allocated at init time
- As process is created, a PCB is assigned and initialized for it
  - Often process id is an offset into an array of PCBs
- After process terminates, PCB is freed (sometimes kept around for parent to retrieve its exit status)

State Queues



Ready queue, queues per device, queue of all processes, …

Context Switch

- When a process is running, some of its state is stored directly in the CPU (register values, etc.)
- When the OS stops a process, it must save all of this hardware state somewhere (PCB) so that it can be restored again
- The act of saving one processes hardware state and restoring another's is called a context switch
  - 100s or 1000s per second!

UNIX process creation

- Fork() system call
  - Creates a new PCB and a new address space
  - Initializes the new address space with a *copy* of the parent's address space
  - Initializes many other resources to copies of the parents (e.g. same open files)
  - Places new process on the queue of runnable processes
- Fork() returns twice: to parent and child
  - Returns child's process ID to the parent
  - Returns 0 to the child

Example Code Snippet

```
int main (int argc, char **argv)
{
   int childPid;
   childPid = fork();
   if (childPid == 0){
       printf(“Child running\n”);
   } else {
       printf(“Parent running: my child is %d\n”,
       childPid);
   }
}
```

Output

```
%./tryfork
Parent running: my child is 707
Child running
%
```

Experiments

- Try putting an infinite loop in the child's portion (do you return to the command shell?) and then looking for it in the ps output
- Try putting an infinite loop in the parent's portion (do you return to the command shell?)
- Put an infinite loop in both
  - try killing the child (look in the ps output for the child and the parent)
  - Try killing the parent - what happens to the child?

Fork and Exec

- How do we get a brand new process not just a copy of the parent?
  - Exec () system call
  - `int exec (char * prog, char ** argv)`
- Exec:
  - Stops the current process
  - Loads the program, prog, into the address space
  - Passes the arguments specified in argv
  - Places the PCB back on the ready queue
- Exec "takes over" the process
  - There is no going back to it when it returns
  - Try to exec something in your shell (example: exec ls) – when ls is done your shell is gone because ls replaced it!

UNIX Shell

```
int main (int argc, char **argv)
{
   while (1){
        int childPid;
        char * cmdLine = readCommandLine();

        if (userChooseExit(cmdLine)){
                        wait for all background jobs
        }

        childPid = fork();
        if (childPid == 0){
                setSTDOUT_STDIN_STDERR(cmdLine);
                exec ( getCommand(cmdLine));

        } else {
                if (runInForeground(cmdLine)){
                        wait (childPid);
                }
        }
}
```

Windows Process Creation

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName, // name of executable module
  LPTSTR lpCommandLine, // command line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
  BOOL bInheritHandles, // handle inheritance option
  DWORD dwCreationFlags, // creation flags
  LPVOID lpEnvironment, // new environment block
  LPCTSTR lpCurrentDirectory, // current directory name
  LPSTARTUPINFO lpStartupInfo, // startup information
  LPPROCESS_INFORMATION lpProcessInformation //
        information );
```

Windows vs Unix

- Windows doesn't maintain the same relationship between parent and child
  - Later versions of Windows have concept of "job" to mirror UNIX notion of parent and children (process groups)
- Waiting for a process to complete?
  - WaitforSingleObject to wait for completion
  - GetExitCodeProcess (will return STILL_ALIVE until process has terminated)

Cooperating Processes

- Processes can run independently of each other or processes can coordinate their activities with other processes
- To cooperate, processes must use OS facilities to communicate
  - One example: parent process waits for child
  - Many others
    - Shared Memory
    - Files
    - Sockets
    - Pipes
    - Signals
    - Events
    - Remote Procedure Call

Sockets

- A socket is an end-point for communication over the network
- Create a socket
  - `int socket(int domain, int type, int protocol)`
  - Type = SOCK_STREAM for TCP
- Read and write socket just like files
- Can be used for communication between two processes on same machine or over the network

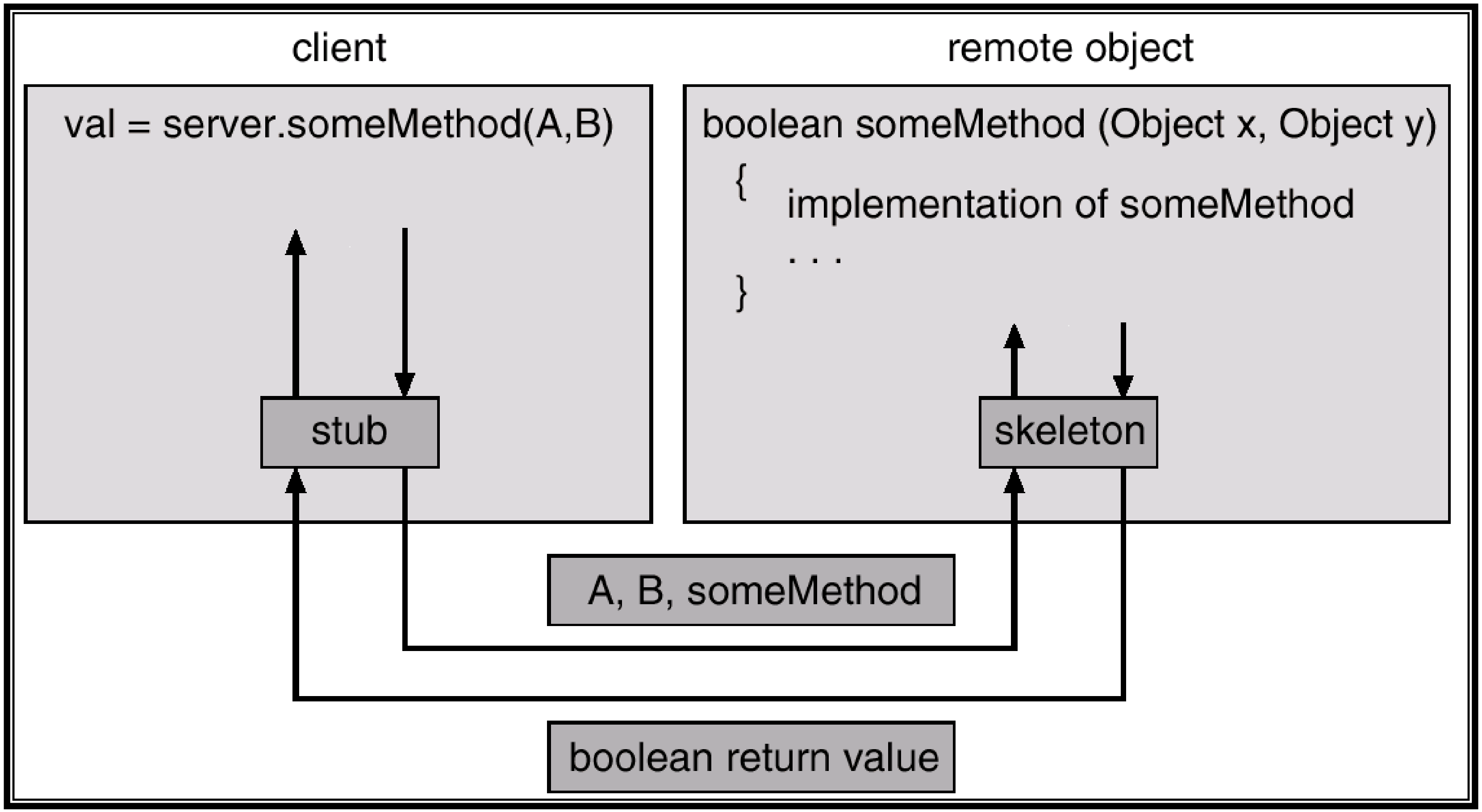
Pipes

- Bi-directional data channel between two processes on the same machine
- Created with:
  - `int pipe (int fildes[2])`
- Read and write like files

Signals

- Processes can register to handle signals with the signal function
  - `void signal (int signum, void (*proc) (int))`
- Processes can send signals with the kill function
  - `kill (pid, signum)`
- System defined signals like SIGHUP (0), SIGKILL (9), SIGSEGV(11)
  - In UNIX shell, try: "kill -9 pidOfProcessYouDon'tReallyCareAbout"
- Signals not used by system like SIGUSR1 and SIGUSR2

Remote Procedure Call (RPC)



Processes

- What is a process?
- Process States
- Switching Between Processes
- Process Creation
- PCBs
- Communication/Cooperation between processes