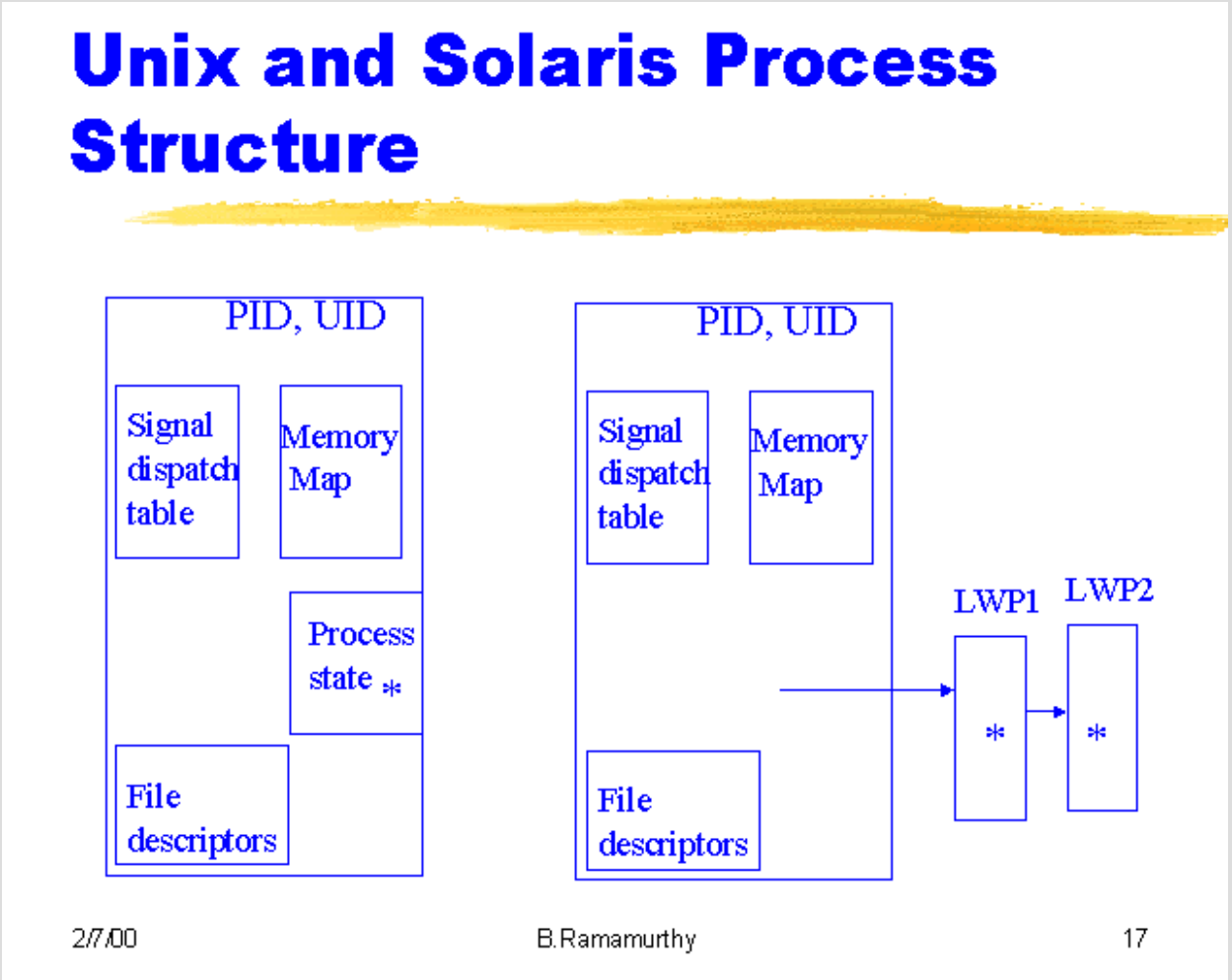
Unix Processes



Picture copied from
http://www.cse.buffalo.edu/~bina/cse421/spring00/lec4/

Unix Processes

Component of a Process

•address space
•memory pages (usually 1Kbytes to 4Kbytes)
•usually virtual memory now - some pages may be on disk, some in memory
•address space contains program code, variables, process stack, other infomation
•data structures in kernel These contain information such as:
•process id (PID) and parent process id (PPID)

- Unique number assigned by kernel. Assigns the next available PID. Treats this as a circular system, that is when it reaches maximum number, starts again at 0.
- Unix creates new process by spawning a copy of an existing (usually a shell or GUI) process using fork and then substituting the text of another program using exec.
- owner and group, real and effective - UID, EUID, GID, EGID. UID and EUID are the same except for setuid programs. When a process is spawned its GID is set to that of parent.
- priority and nice value. When the kernel schedules processes, the one with highest internal priority is chosen. The is calculated from nice value, the CPU time consumed, and time the process has been ready (waiting to run). The nice value can be set with nice.
- controlling terminal
- process group
- current process state
- process address space map
- resources used
- signal mask

Unix Processes

Life Cycle of a Process:

Unix creates new process by spawning a copy of an existing process using fork and then substituting the text of another program using exec. In UNIX, all processes such as a shell or GUI creates, are children spawned by init, PID 1. The shell or GUI directly or indirectly spawn users processes.

Init plays another important role. Exiting processes call _exit and return an exit status. This is stored by the kernel until requested by the process's parent (using the wait system call). The address space of the process is released and it uses no further CPU time. However it still retains its identity (PID). This uses a slot in the process table but no other resources.

Problems arise if

- parent fails to call wait
- parent dies first - in this case init becomes the parent of the process and waits on it. (init waits on all orphans.) Sometimes this does not happen and the processes remain in the system as zombies. These appear with status Z or exiting.
-

Processes

- Monitoring Process
  - ps
  - pstree
  - parent process
  - child process
- exec, execvp
- fork

exec

EXEC(3) Linux Programmer's Manual EXEC(3)

NAME

execl, execlp, execle, execv, execvp - execute a file

SYNOPSIS

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

How does a program execute a program

- program reads command line
- program calls execvp
- main()

```
{    char   *arglist[4];
arglist[0] = "ls";
arglist[1] = "-l";
arglist[2] = "/bin";
arglist[3] = 0 ;
     execvp( "ls" , arglist );
     printf("ls is done. bye\n");}
```

Fork

NAME

fork - create a child process

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

DESCRIPTION

fork creates a child process that differs from the parent process only in its PID and PPID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited.

Under Linux, fork is implemented using copy-on-write pages, so the only penalty incurred by fork is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.

RETURN VALUE

On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and errno will be set appropriately.

getpid

NAME

getpid, getppid - get process identification

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

DESCRIPTION

getpid returns the process ID of the current process. (This is often used by routines that generate unique temporary file names.)

getppid returns the process ID of the parent of the current process.

fork

- What does this fragment do?
  printf(“mypid=%d\n”,getpid());
  n=fork();
  printf(“mypid=%d,n=%d\n”,getpid(),n);
- What does this fragment do?
  printf(“mypid=%d\n”,getpid());
  fork();
  fork();
  fork();
  printf(“mypid=%d\n”,getpid());

sleeping while child works:wait

NAME

wait - wait for process termination

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

DESCRIPTION

The wait function suspends execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed.

sleeping while child works:wait

NAME

waitpid - wait for process termination

SYNOPSIS

#include <sys/types.h>

#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);

DESCRIPTION

The waitpid function suspends execution of the current process until a child as specified by the pid argument has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child as requested by pid has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed.

sleeping while child works:wait

The value of options is an OR of zero or more of the following constants:

WNOHANG
which means to return immediately if no child has exited.

WUNTRACED
which means to also return for children which are stopped (but not traced), and whose status has not been reported. Status for traced children which are stopped is provided also without this option.

:

sleeping while child works:wait

The value of pid can be one of

< -1 which means to wait for any child process whose process group ID is equal to the absolute value of pid.

-1 which means to wait for any child process; this is the same behaviour which wait exhibits.

0 which means to wait for any child process whose process group ID is equal to that of the calling process.

>0 which means to wait for the child whose process ID is equal to the value of pid

Components of a Process, /proc

PROC(5) Linux Programmer's Manual PROC(5)

NAME

proc - process information pseudo-filesystem

DESCRIPTION

The proc filesystem is a pseudo-filesystem which is used as an interface to kernel data structures. It is commonly mounted at /proc. Most of it is read-only, but some files allow kernel variables to be changed.

Components of a Process, /proc

/proc/[number]/stat

Status information about the process. This is used by ps(1). It is defined in /usr/src/linux/fs/proc/array.c. The fields, in order, with their proper scanf(3) format specifiers, are:

pid %d The process id.

comm %s

The filename of the executable, in parentheses. This is visible whether or not the executable is swapped out.

state %c

One character from the string "RSDZTW" where R is running, S is sleeping in an interruptible wait, D is waiting in uninterruptible disk sleep, Z is zombie, T is traced or stopped (on a signal), and W is paging.

ppid %d

The PID of the parent.

pgrp %d

The process group ID of the process.

session %d

The session ID of the process.

tty_nr %d

The tty the process uses.

tpgid %d

process info system calls

chdir(path)
fchdir(fd)
getcwd()
ctermid()
Return the filename corresponding to the controlling terminal of the process.
getegid()
Return the effective group id of the current process. This corresponds to the `set id' bit on the file being executed in the current process.
geteuid()
Return the current process' effective user id.

process info system calls

getgid()
Return the real group id of the current process.
getgroups()
Return list of supplemental group ids associated with the current process.
getlogin()
Return the name of the user logged in on the controlling terminal of the process.
getpgid(pid)
Return the process group id of the process with process id pid.
getpgrp()
Return the id of the current process group.

process info system calls

getpid()
Return the current process id.
getppid()
Return the parent's process id..
getuid()
Return the current process' user id..
getenv(varname)
Return the value of the environment variable varname.
putenv(string)
Set the environment variable as defined by string.

process info system calls

setegid(egid)
Set the current process' effective group id.
seteuid(euid)
Set the current process's effective user id. Availability: Unix.
setgid(gid)
Set the current process' group id. Availability: Unix.
setgroups(size, list)
Set the list of supplemental group ids associated with the current process to whose store in list
setpgrp(pid, pgid)
Sets the process group ID of the process specified by pid to pgid.

process info system calls

setreuid(ruid, euid)
Set the current process's real and effective user ids.
setregid(gid, egid)
Set the current process's real and effective group ids.
getsid(pid)
Returns the session ID of the calling process for process pif.
setsid()
Run a program in a new session
setuid(uid)
Set the current process' user id.

Environ

```
#include <unistd.h>
extern char **environ;
int main(int agrc, char **argv){
  char **a;
  a=environ;
  printf("Starting myenviron, my environment variables  are:\n");
    while (*a){
      printf("\t%s\n",*a);
      a++;
    }
}
```

sysinfo

NAME

sysinfo - returns information on overall system statistics

SYNOPSIS

#include <sys/sysinfo.h>

int sysinfo(struct sysinfo *info);

DESCRIPTION

Until Linux 2.3.16, sysinfo used to return information in the following structure:

sysconf

NAME

sysconf - Get configuration information at runtime

SYNOPSIS

#include <unistd.h>

long sysconf(int name);

DESCRIPTION

POSIX allows an application to test at compile- or run-time whether certain options are supported, or what the value is of certain configurable constants or limits.

Studying Windows Internals

Look at www.sysinternals.com

- Process Explorer
- Handle
- ListDLLs
- Pstools
- Regmon

OVERVIEW

- A Windows process contains its own independent virtual address space with both code and data
- Each process contains one or more independently executed threads
- The Windows thread is the basic executable unit
- A process can
  - Create new threads within the processes
  - Create new, independent processes
  - Manage communication and synchronization between these objects
- For now we consider only a single thread within a process

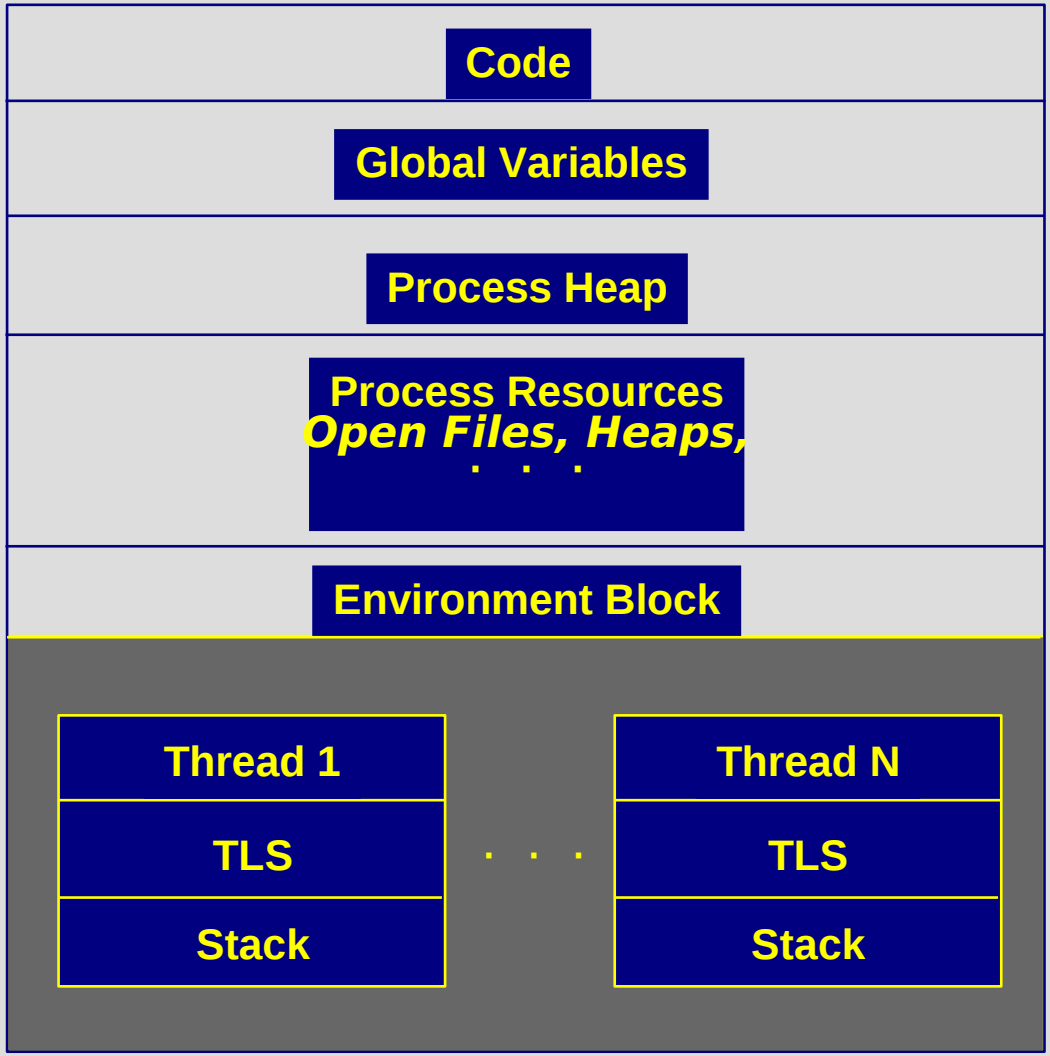
Windows PROCESSES HAVE

- One or more threads
- Virtual address space which is distinct from other processes' address spaces
  - Except for shared memory-mapped files
- One or more code segments
- One or more data segments containing global variables
- Environment strings with environment variable information
- The process heap
- Resources such as open handles and other heaps

THREADS

- Share the code, global variables, environment strings and resources in a process
- Are independently scheduled
- Have a stack for procedure calls, interrupts, etc.
- Have Thread Local Storage (TLS)—pointers giving each thread the ability to allocate storage to create its own unique data environment
- Have an argument (on the stack) from the creating thread
  - Can also be unique for each thread
- Have a context structure, maintained by the kernel, with machine register values

A PROCESS AND ITS THREADS



PROCESS CREATION

```
BOOL CreateProcess (LPCTSTR lpImageName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpsaProcess,
  LPSECURITY_ATTRIBUTES lpsaThread,
  BOOL bInheritHandles, DWORD dwCreate,
  LPVOID lpvEnvironment, LPCTSTR lpCurDir,
  LPSTARTUPINFO lpsiStartInfo,
  LPPROCESS_INFORMATION lppiProcInfo)
```

Return: TRUE only if the process and thread are successfully created

PROCESS CREATION

Parameters

`lpImageName` — Specifies the executable program

`lpCommandLine` — Specifies the command line arguments

`lpsaProcess` — Points to the process security attribute structure

`lpsaThread` — Points to the thread security attribute structure

- (`NULL` implies default security)

PROCESS CREATION

`bInheritHandles` — This is a “master switch” to indicate that the new process can inherit handles from the parent

- Individual handles must still be specified as inheritable
- A typical use is to redirect standard I/O — there are numerous examples in the lab exercises

PROCESS CREATION

`dwCreate` — Combines flags, including:

- `CREATE_SUSPENDED` — The primary thread is in a suspended state and will only run when `ResumeThread` is called
- `DETACHED_PROCESS` — Creates a process without a console
- `CREATE_NEW_CONSOLE` — Gives the new process a console
- (These two are mutually exclusive. If neither is set, the process inherits the parent's console.)
- `CREATE_NEW_PROCESS_GROUP` — Specifies that the new process is the root of a new process group

PROCESS CREATION

`lpvEnvironment` — Points to an environment block for the new process. If `NULL`, the parent's environment is used. Contains name/value strings, such as search path.

`lpCurDir` — Drive and directory for the new process

- If `NULL`, parent's is used)

`lpsiStartInfo` — Main window appearance for the new process

`lppiProcInfo` — Structure to contain the returned process and thread handles and identification

PROCESS CREATION

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
  } PROCESS_INFORMATION;
```

Processes and threads need both handles and IDs

- ID is unique to the object for its lifetime in all processes
- There may be several handles for a given process
- Handles are used with many general-purpose functions

PROCESS CREATION

```
typedef struct _STARTUPINFO {
/* Lots of information controlling the window
  */
  DWORD dwFlags;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFO;
```

- Setting these handles before creating the process is a common way to redirect the new process' standard I/O
- Set `dwFlags` to `STARTF_USESTDHANDLES` to enable redirection
- Use `GetStartupInfo (&StartupInfo)` to fill in the structure from the parent's values

PROCESS CREATION

`lpImageName` and `lpCommandLine` combine to form the executable image name. The rules for determining the command line are:

- `lpImageName`, if not `NULL`, is the name of the executable
- Otherwise, executable is the first token in `lpCommandLine`

A process written in C can obtain command line entries with `argc/argv`

There is a `GetCommandLine` function

PROCESS CREATION

Rules for `lpImageName`:

- If `lpImageName` is not `NULL`, it specifies the executable module. Use full path name, or use a partial name and the current drive and directory will be used. You must include the file extension.

- If `lpImageName` is `NULL`, the first white-space delimited token in `lpCommandLine` is the program name. If no extension is specified, `.EXE` is assumed.

PROCESS CREATION

If the name does not contain a full directory path, the search sequence is:

- The directory of the current process' image
- The current directory
- The Windows system directory, which you can retrieve with `GetSystemDirectory`
- The Windows directory, which you can retrieve with `GetWindowsDirectory`
- The directories as specified in the environment variable `PATH`

PROCESS CREATION

`HANDLE GetCurrentProcess (VOID)`

- Return: a “pseudo handle” which is not inheritable
- Can be used whenever a process needs its own handle

`DWORD GetCurrentProcessId (VOID)`

- Return: The process ID of the current process

PROCESS CREATION

```
typedef struct SECURITY_ATTRIBUTES {
      DWORD nLength;
      LPVOID lpSecurityDescriptor;
      BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

The `bInheritHandle` flag determines whether the child processes can inherit this specific handle

- By default, a handle is not inheritable
- `bInheritHandle` should be set to `TRUE`
- Parent communicates inheritable handle values to child with IPC or by assigning an handle to standard I/O
  - Typically in the `STARTUPINFO` structure