Processes lecture 3

The end of a process - 4.3.2

UNIX state transitions

Context switching - 4.2.3

Process termination

All resources must be accounted for

- may be found in the PCB or other tables must be searched
- e.g. devices, memory, files
- just reduce usage count on shared resources
  - memory, libraries, files/buffers
  - (can this shared library be released from memory now?)
- if the process doesn't tidy up e.g. close files, then something else must
- accounting information is updated
- was this a login process?
  - remove the user from the system
- notify the relatives?

UNIX stopping

Usually call *exit(termination status)*

open files are closed - including devices

memory is freed

accounting updated

state becomes "zombie"

children get "init" as a step-parent

parent is signalled (in case it is waiting or will wait)

after the parent retrieves the termination status the PCB is freed

Two reasons to stop

by completing normally

forcibly stopped by another process or the OS

Stopping normally

must call an exit routine

this does all the required tidying up

what if it doesn't call exit and just doesn't have a next instruction?

Forced stops

- only certain processes can stop others
  - parents
  - owned by the same person
  - same process group
- why do they do it?
  - work no longer needed
  - somehow gone wrong
- OS also stops processes
  - usually when something has gone wrong
  - exceeded time
  - tried to access some prohibited resource

Cascading termination

Some systems don't allow child processes to continue when the parent stops.

UNIX process state diagram

Context switching

What is the context?

registers

memory - including dynamic elements such as call stack

files, resources

but also things like caches, TLBs - these are normally lost

The context changes as the process executes.

Leaving user level code to enter the kernel

This changes the context - some people refer to this as a context switch as well.

Normally “context switch” means the change from one process running to another.

Three ways to change context from user level to system level

System calls

Exceptions

Interrupts

System calls

special instruction to change the processor's mode and jump to a predefined address

system call - supervisor call - change mode to kernel

We saw the need for two modes for the processor to operate in earlier in the course.

problem/supervisor, user/kernel

System call interface

Usually there are not different entry points for each system call.

Go to the same place and a table is used to pass the call to the correct handler.

So the system call instruction e.g. chmk, needs information as to which system call

Frequently over 100 system calls in an OS.

The information as to which one can be passed as a parameter - maybe placed on a kernel stack by the instruction itself.

Passing parameters into system calls

registers

with actual parameters or a pointer to the parameters

stack

just like an ordinary function call

It may require some memory mapping trickery to access the address space of the process.

In monolithic kernels it is common to have the kernel shared across the page tables of all processes.

Some architectures make this almost compulsory.

The top half of memory is commonly reserved for the OS. A lot of 32-bit systems allow 2 gigabytes of virtual address for their processes.

Care with checking the parameters

Especially pointers to chunks of memory.

e.g.

write(toFile, fromHere)

what check needs to be made on fromHere?

The process can read from it. Because if it can't it is just putting the output into a file which it can now read from.

and

read(fromFile, toHere)

what check needs to be made on toHere?

The process can write to it. Otherwise the OS will dump on some memory which is not under the control of this process.

Getting the results

registers

deposit on the calling stack

directly into a user level data structure

UNIX system calls

Most system calls are hidden away inside user level library routines

(The C library has over 300 functions which call about 150 different real system calls)

This means the call from within the program is just a normal function call.

Parameters pushed on the stack.

Possibly some library level work before…

System call instruction is performed

now in kernel mode at the system call interface

previous context is saved

which system call

move the parameters to the kernel stack

ordinary jump to the code to handle this call

check the parameters and whether the process has the correct permissions

if ok perform the system call
if not or if errors start handling the errors
else
the saved context of the process may be altered
    this is one way returning results
return from the system call instruction (e.g. rei)
    back to library code
possibly handle errors
normal return to the original call

Exceptions

Process/thread does something wrong e.g.
    Divide by zero
    access invalid memory
    attempt to perform illegal instruction
Causes a trap to a particular OS function to handle
Possibly pass the exception back to the

Interrupts

Special case of the clock interrupt
How much time has this process had - in user/kernel modes?
Updates the time values for the process.