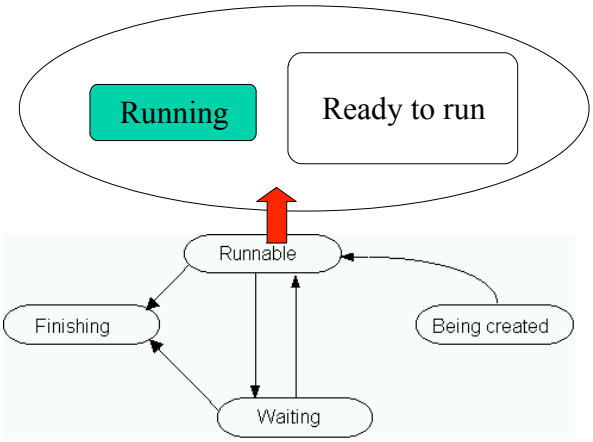
Runnable

On a single processor only one process/thread can run at a time.
Many may however be runnable - either *running* or *ready to run*.



Preemptive multitasking

A clock interrupt causes the OS to check to see if the current thread should continue
Each thread has a time slice
How is the time slice allocated?

What advantages/disadvantages does preemptive multitasking have over cooperative multitasking?

Advantages

- control
- predictability

Disadvantages

- critical sections
- efficiency

Cooperative multitasking

Two main approaches

1. a process yields its right to run
2. system stops a process when it makes a system call

This does **NOT** mean a task will work to completion without allowing another process to run. e.g. Macintosh before OS X and early versions of Windows

A mixture

Older versions of UNIX (including versions of Linux before 2.6) have not allowed preemptive multitasking when a process has made a system call.

Context switch

The change from one process running to another one running on the same processor is usually referred to as a "context switch".

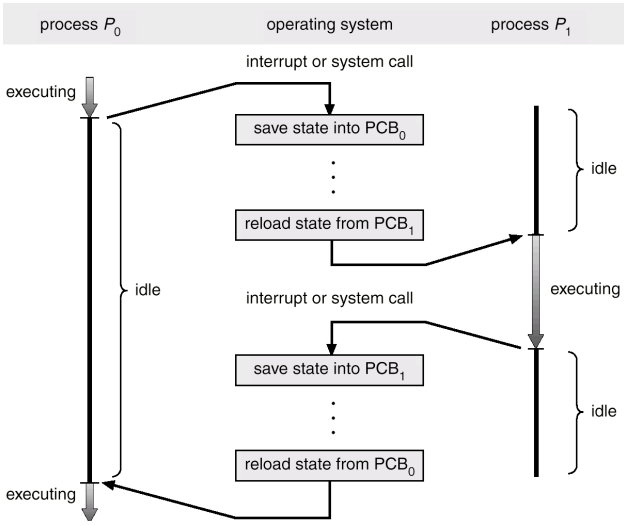
What is the context?

- registers
- memory - including dynamic elements such as the call stack
- files, resources
- but also things like caches, TLB values - these are normally lost

The context changes as the process executes.

But normally a "context switch" means the change from one process running to another, or from a process running to handling an interrupt. Whenever the process state has to be stored and restored.

Context switch (cont.)



Returning to running

State transition

- Must store process properties so that it can restart where it was.
- If changing processes the page table needs altering.
- Rest of environment must be restored.
- If changing threads within the same process simply restoring registers might be enough.

Some systems have multiple sets of registers which means that a thread change can be done with a single instruction.

Waiting

Processes seldom have all the resources they need when they start

- memory
- data from files or devices e.g. keyboard input

Waiting processes must not be allowed to unnecessarily consume resources, in particular the processor.

- state is changed to waiting

may be more than one type of waiting state

short wait e.g. for memory

long wait e.g. for an archived file (see suspended below)

- removed from the ready queue
- probably entered on a queue for whatever it is waiting for

when the resource becomes available

- state is changed to runnable
- removed from the waiting queue
- put back on the runnable queue

Suspended

Another type of waiting

ctrl-z in some UNIX shells

Operators or OS temporarily stopping a process – i.e. it is not (usually) caused by the process itself

- allows others to run to completion more rapidly
- or to preserve the work done if there is a system problem
- or to allow the user to restart the process in the background etc.

Suspended processes are commonly swapped out of real memory.

This is one state which affects the process, individual threads aren't swapped out. Why not?

See `infinite.c` or `infinite.py` and use ctrl-z, then do `ps`

to resume you type `fg` (foreground), also play with the `jobs` command

ctrl-z sends the same signal as

```
os.kill(pid, signal.SIGSTOP)
```

Why we don't use Java suspend()

If dealing with threads in Java we don't use these deprecated methods:

`suspend()` freezes a thread for a while. This can be really useful.

`resume()` releases the thread and it can start running again.

But we can *easily(?)* get deadlock.

`suspend()` keeps hold of all locks gathered by the thread.

If the thread which was going to call `resume()` needs one of those locks before it can proceed we get stuck.

Java threads and "stop"

Why we don't use stop()

`stop()` kills a thread forcing it to release any locks it might have.

We will see where those locks come from in later lectures.

The idea of using locks is to protect some shared data being manipulated simultaneously.

If we use `stop()` the data may be left in an inconsistent state when a new thread accesses it.

Waiting in UNIX

A process waiting is placed on a queue.

The queue is associated with the hash value of a kernel address

(waiting or suspended processes may be swapped out)

when the resource becomes available

- originally used to scan whole process table
- all things waiting for that resource are woken up
- (may need to swap the process back in)
- first one to run gets it
- if not available when a process runs the process goes back to waiting

a little like in Java

```
while (notAvailable)
  wait();
```


Finishing

All resources must be accounted for

may be found in the PCB or other tables
e.g. devices, memory, files

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

remove any associated processes

Was this a session leader? If so then should all processes in the same session be removed?

remove the user from the system

notify the relatives?

Two reasons to stop

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
- user got bored waiting for completion

OS also stops processes

- usually when something has gone wrong
- exceeded time
- tried to access some prohibited resource

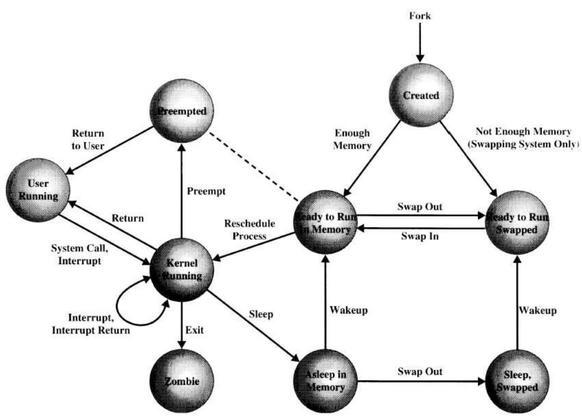
<u>Cascading termination</u>

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

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

UNIX state diagrams



Info from a Linux process table

`ps -e -o s,uid,pid,ppid,group,sess,c,pri,ni,rss,sz:7,wchan:30,tty,time,cmd`

```
S   UID   PID  PPID GROUP     SESS  C PRI  NI   RSS      SZ WCHAN                          TT           TIME CMD
S     0     1     0 root         1  0  19   0  2684    6700 poll_schedule_timeout          ?        00:00:01 /sbin/init
S     0     2     0 root         0  0  19   0     0       0 kthreadd                       ?        00:00:00 [kthreadd]
S     0     3     2 root         0  0  19   0     0       0 smpboot_thread_fn              ?        00:00:00 [ksoftirqd/0]
S     0     5     2 root         0  0  39 -20     0       0 worker_thread                  ?        00:00:00 [kworker/0:0H]
S     0     7     2 root         0  0  39 -20     0       0 worker_thread                  ?        00:00:00 [kworker/u:0H]
S     0     8     2 root         0  0 139   -     0       0 cpu_stopper_thread             ?        00:00:00 [migration/0]
S     0     9     2 root         0  0  19   0     0       0 rcu_gp_kthread                 ?        00:00:00 [rcu_bh]
S     0    10     2 root         0  0  19   0     0       0 rcu_gp_kthread                 ?        00:00:00 [rcu_sched]
S     0    11     2 root         0  0 139   -     0       0 smpboot_thread_fn              ?        00:00:00 [watchdog/0]
S     0    12     2 root         0  0 139   -     0       0 smpboot_thread_fn              ?        00:00:00 [watchdog/1]
S     0    13     2 root         0  0  19   0     0       0 smpboot_thread_fn              ?        00:00:00 [ksoftirqd/1]
S     0    14     2 root         0  0 139   -     0       0 cpu_stopper_thread             ?        00:00:00 [migration/1]
S     0    16     2 root         0  0  39 -20     0       0 worker_thread                  ?        00:00:00 [kworker/1:0H]
S     0    17     2 root         0  0  39 -20     0       0 rescuer_thread                 ?        00:00:00 [cpuset]
S     0    18     2 root         0  0  39 -20     0       0 rescuer_thread                 ?        00:00:00 [khelper]
... lots of processes removed
S  1000  2098  1348 robert    1348  0  19   0 23636   97269 poll_schedule_timeout          ?        00:00:00 gnome-screensaver
S  1000  2121  1348 robert    1348  0  19   0 13932  129253 poll_schedule_timeout          ?        00:00:00 update-notifier
S  1000  2139     1 robert    1348  0   9  10 126180 160715 poll_schedule_timeout          ?        00:00:39 /usr/bin/python3 /usr/b
S  1000  2174     1 robert    1403  0  19   0  3760   69016 poll_schedule_timeout          ?        00:00:00 /usr/lib/libunity-webap
S  1000  2268     1 robert    1403  0  19   0  8988  253634 poll_schedule_timeout          ?        00:00:00 /usr/lib/gvfs/gvfsd-htt
S  1000  2278  2268 robert    1403  0  19   0   628    1110 wait                           ?        00:00:00 sh -c /usr/lib/x86_64-l
S  1000  2279  2278 robert    1403  0  19   0  3412   49887 poll_schedule_timeout          ?        00:00:00 /usr/lib/x86_64-linux-g
S  1000  2305  1348 robert    1348  0  19   0  4844  111973 poll_schedule_timeout          ?        00:00:00 /usr/lib/x86_64-linux-g
S  1000  2364     1 robert    1403  0  19   0  6068   60692 poll_schedule_timeout          ?        00:00:00 /usr/lib/geoclue/geoclu
S  1000  2368     1 robert    1403  0  19   0  7744   86212 poll_schedule_timeout          ?        00:00:00 /usr/lib/ubuntu-geoip/u
S  1000  2371  2368 robert    1403  0  19   0   624    1110 wait                           ?        00:00:00 sh -c /usr/lib/x86_64-l
S  1000  2372  2371 robert    1403  0  19   0  3412   49887 poll_schedule_timeout          ?        00:00:00 /usr/lib/x86_64-linux-g
S  1000  2917     1 robert    1348  0  19   0 25208  153503 poll_schedule_timeout          ?        00:00:11 gnome-terminal
S  1000  2925  2917 utmp      1348  0  19   0   844    3708 unix_stream_recvmsg            ?        00:00:00 gnome-pty-helper
S  1000  2926  2917 robert    2926  0  19   0  3016    6520 wait                           pts/2    00:00:00 bash
S  1000  2977  2926 robert    2926  0  19   0  1524    5582 wait                           pts/2    00:00:00 man ps
S  1000  2987  2977 robert    2926  0  19   0   992    3443 n_tty_read                     pts/2    00:00:00 pager -s
S  1000  2993  2917 robert    2993  0  19   0  3104    6520 wait                           pts/3    00:00:00 bash
S     0  3565     2 root         0  0  19   0     0       0 worker_thread                  ?        00:00:00 [kworker/1:2]
S     0  3671     2 root         0  0  19   0     0       0 worker_thread                  ?        00:00:00 [kworker/0:1]
R  1000  3688  2993 robert    2993  0  19   0  1300    5661 -                              pts/3    00:00:00 ps -e -o s,uid,pid,ppid
```

Before next time

Read from the textbook

6.1 Basic Concepts
6.2 Scheduling Criteria
6.3 Scheduling Algorithms
6.5 Multiple-Processor Scheduling