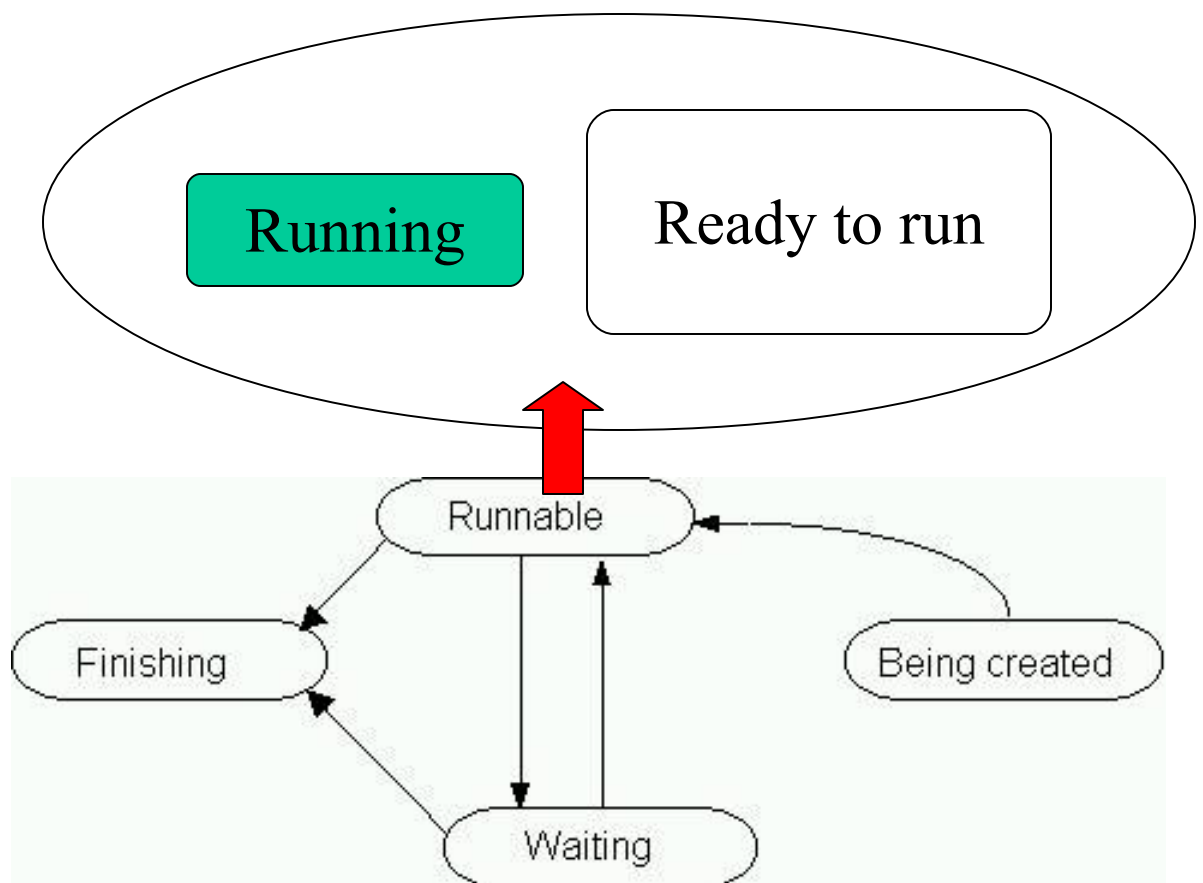


Runnable

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 process/thread should continue

Each process/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 some versions of Windows

A mixture

Older versions of UNIX (including recent versions of Linux) 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". Sometimes we also talk about a context switch when a process performs a system call.

What is the context?

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

The context changes as the process executes.

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

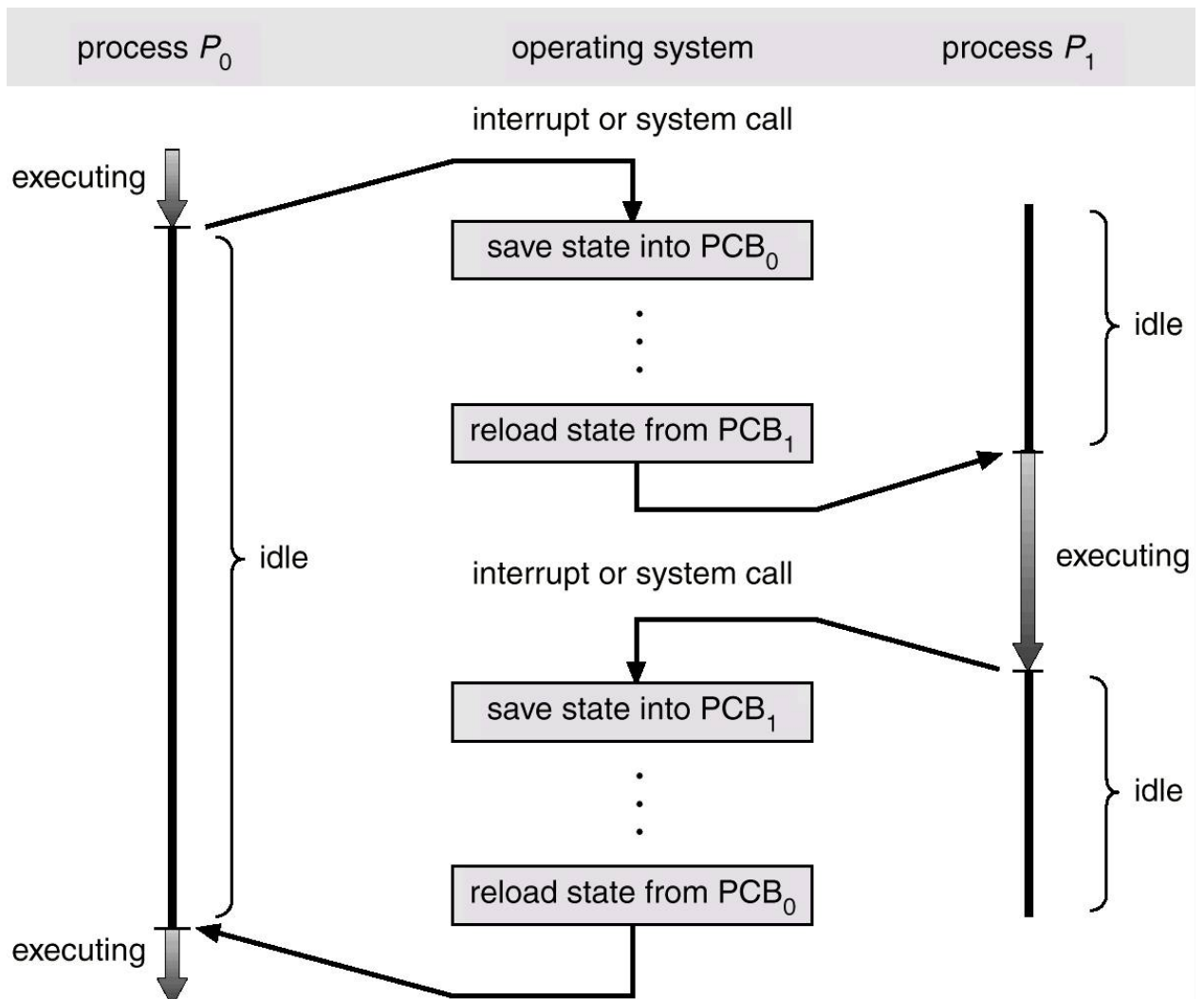
Returning to running

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 could be done with a single instruction.

Context switch (cont.)



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 vs
 - 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

- allows others to run to completion more rapidly
- or to preserve the work done if there is a system problem

Suspended processes are commonly swapped out of real memory.

This is one state which affects the process, individual threads aren't suspended (in the sense of being able to be swapped out). Why not?

Why we don't use Java `suspend()`

`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.

Ruby equivalents are `stop` and `run`. But Ruby hasn't deprecated them.

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.

Ruby has `thread.kill`, which is almost as bad, but Ruby can stop other threads running really easily:

```
Thread.critical=true
```

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 must be searched

- 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
was this a session leader?

remove any associated processes
was this a login process?

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

OS also stops processes

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

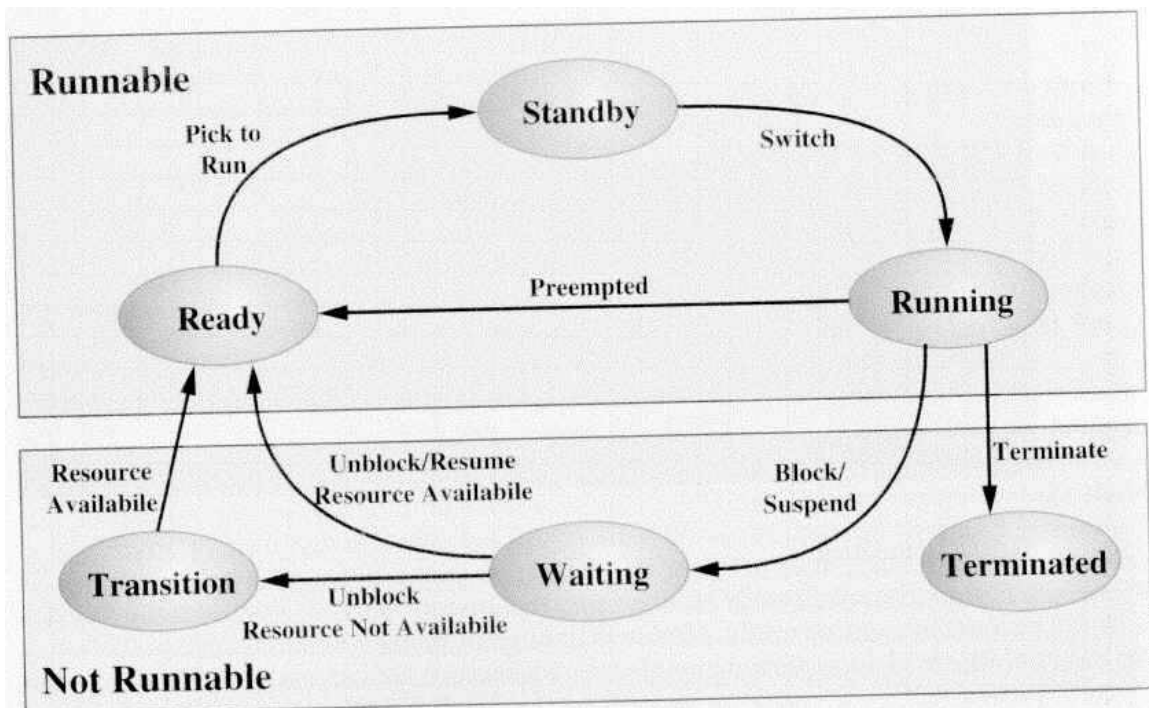
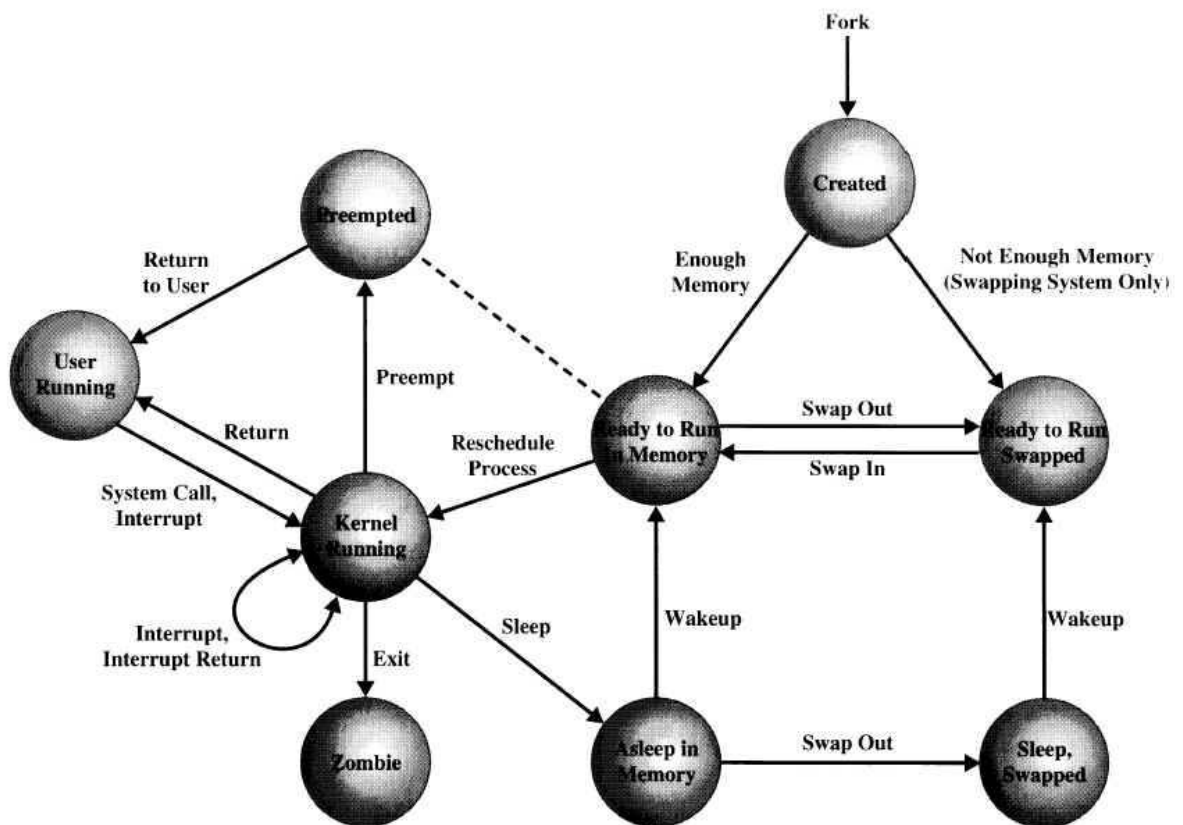
Cascading termination

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

UNIX 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 and NT state diagrams



Info from the process table

I	F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
	3	R	<	0	0	0.0	38	-6	deccc0	146M	*	??	5:23.54	kernel idle
80048001	IL		0	1	0	0.0	44	0	75740	96K	pause	??	0:03.01	init
8001	I		0	3	1	0.0	44	0	e69740	1.0M	sv_msg_	??	0:00.05	kloadsrv
8001	S		0	5	1	0.0	44	0	e9e240	368K	*	??	0:00.00	hotswapd
8001	S		0	307	1	0.0	44	0	b3c240	248K	event	??	0:00.64	syslogd
8001	I		0	311	1	0.0	44	0	9e7740	304K	event	??	0:00.13	binlogd
8001	I		0	386	1	0.0	44	0	b10240	160K	event	??	0:00.02	portmap
8001	I		0	394	1	0.0	44	0	abacc0	120K	event	??	0:00.00	yppbind
8001	S		0	414	1	0.0	44	0	cacc0	80K	event	??	0:00.00	mount
8001	S	<	0	474	1	0.0	0	-44	456240	312K	event	??	0:00.43	xntpd
8001	S		0	506	1	0.0	44	0	48e240	168K	event	??	0:05.72	snmpd
8001	I		0	607	1	0.0	44	0	570240	128K	pause	??	0:00.00	inetd
8001	I		0	608	607	0.0	44	0	4a0240	192K	event	??	0:00.21	inetd
8001	I		0	619	1	0.0	44	0	e30240	264K	1231e99c	??	0:01.08	cron
8001	I		0	692	1	0.0	44	0	e30cc0	296K	event	??	0:00.02	lpd
80808001	S		0	4865	1	0.0	44	0	756240	4.6M	*	??	0:00.88	java
8001	S		0	4908	1	0.0	44	0	8a4cc0	12M	*	??	0:03.18	smsd
80008001	I		0	126818	608	0.0	44	0	e70cc0	296K	event	??	0:00.05	telnetd
80008001	I		0	126922	608	0.0	44	0	646240	304K	event	??	0:00.03	rlogind
80008001	S		0	127087	608	0.0	44	0	a0b740	296K	event	??	0:00.04	telnetd
80008001	I		0	127892	1	0.0	44	0	446240	728K	event	??	0:00.04	xterm
80008001	I		0	127930	608	0.0	44	0	d20240	784K	event	??	0:00.08	xterm
80008001	S		0	128000	608	0.0	44	0	211740	760K	event	??	0:00.21	xterm
80808001	S		22026	128001	128000	0.0	44	0	b1a240	488K	wait	pts/16	0:00.10	bash
82808001	R	+	0	128228	128001	0.0	44	0	446cc0	7.8M	-	pts/16	0:00.05	ps

Before next time

Read from the textbook

4.3 Operations on Processes

4.2 Process Scheduling