-rw-rw----

^^^

| ^^^

| | ^^^

| | |

| | other (world)

| |

| group

|

user

\* fd C++ Java C

\* 0 stdin cin System.in scanf(), read(), getchar()

\* 1 stdout cout System.out printf(), write()

\* 2 stderr cerr System.err fprintf( stderr, "..." ), write()

\* 3 argv[1] (file open for reading or writing)

\* fd 1 stdout is buffered

\* fd 2 stderr is NOT buffered

sprintf( buffer, "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );

write( 1, buffer, 10 ); stdout immediately

printf( "ABCD" );

fprintf( stderr, "ERROR!" );

printf( "EFGH\n" );

**ABCDEFGHIJERROR!ABCDEFGH**

close( 1 ); /\* close stdout \*/

printf( "hmmm fd 1 stdout has been closed\n" );

int fd = open( argv[1], O\_WRONLY | O\_CREAT | O\_TRUNC, 0660 );

printf( "where does this output go?!\n" );

fflush( NULL );

**To a file:**

**hmmm fd 1 stdout has been closed**

**where does this output go?**

Fork bomb! (don't try this at home....)

while ( 1 )

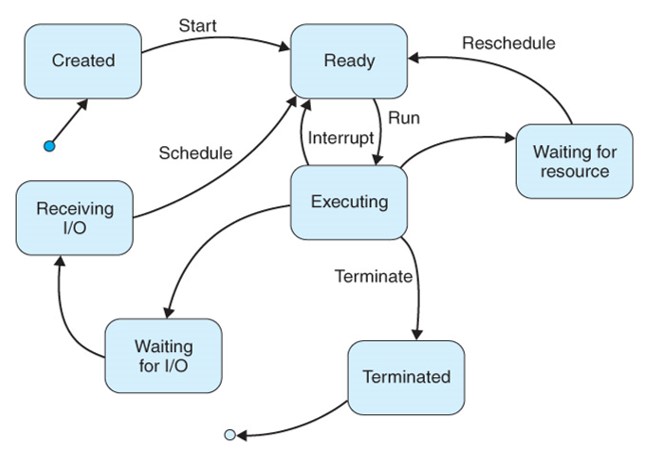
{

fork();

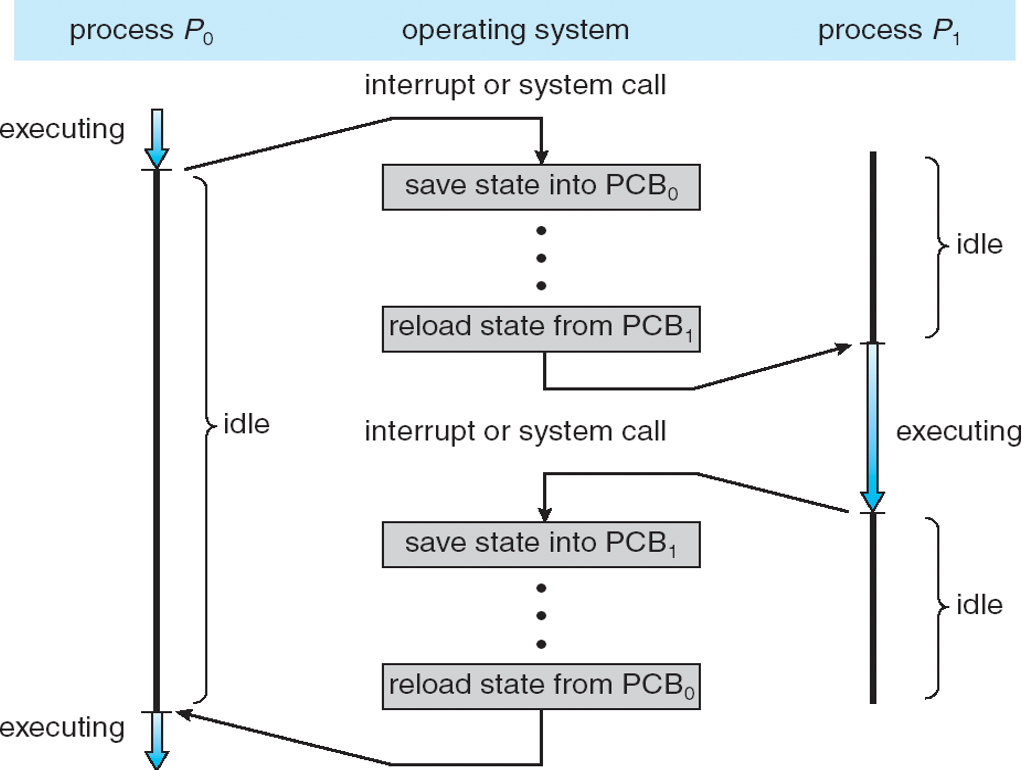
}

**Die soon**

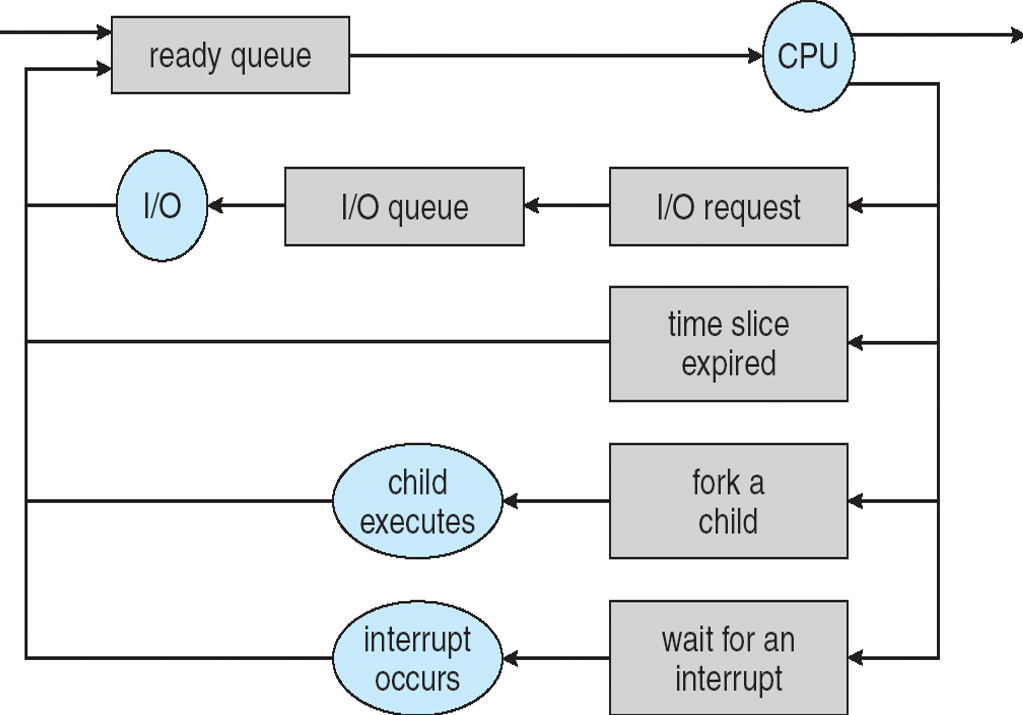
process-states



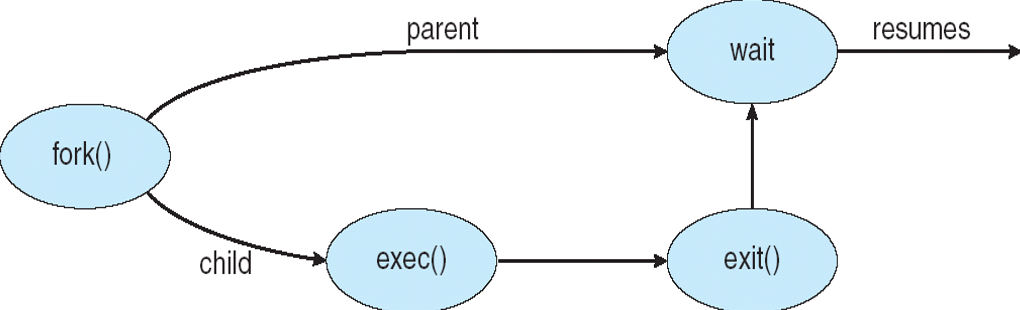
pcb-context-switch



cpu-scheduling



fork-diagram



int x = 5;

pid\_t pid; /\* process id (int) \*/

pid = fork(); /\* create a child process \*/

if ( pid == 0 ) /\* child process \*/

{

printf( "CHILD: happy birthday!\n" );

x += 10;

printf( "CHILD: x is %d\n", x );

printf( "CHILD: bye\n" );

sleep( 5 );

return 12;

}

else /\* pid > 0 so we're in the parent (calling) process \*/

{

printf( "PARENT: my child's pid is %d\n", pid );

x += 30;

printf( "PARENT: x is %d\n", x );

/\* wait for the child to terminate \*/

int status;

pid\_t child\_pid = wait( &status );

/\* wait() blocks indefinitely ... \*/

printf( "PARENT: child %d terminated...", (int)child\_pid );

if ( WIFSIGNALED( status ) ) // core dump or kill or kill -9

{

printf( "abnormally\n" );

}

else if ( WIFEXITED( status ) ) // child called return or exit()

{

int rc = WEXITSTATUS( status );

printf( "successfully with exit status %d\n", rc );

}

printf( "PARENT: bye\n" );

}

When a parent process terminates before its child process, the child process is inherited by process ID 1

Process ID 1 has the responsibility of acknowledging that the child has terminated

REQUIRED: for every process that terminates, its termination MUST be acknowledged (i.e. via wait() system call)

If a process is NOT acknowledged, then the process is a **zombie** (defunct) process

NOTE! The fd table in the parent is inherited by the child process

(very useful for pipes....)

/\* fd table:

0: stdin

1: stdout

2: stderr

3: p[0] <=====read========

4: p[1] ======write=======> \*/

int p[2]; /\* array to hold the two pipe file descriptors

(p[0] is the read end; p[1] is the write end) \*/

int rc = pipe( p );

if ( rc < 0 )

{

perror( "pipe() failed" );

return EXIT\_FAILURE;

}

int bytes\_written = write( p[1], "ABCDEFGHIJKLMNOPQRSTUV", 12 );

printf( "Wrote %d bytes\n", bytes\_written );

char buffer[10];

int bytes\_read = read( p[0], buffer, 6 ); /\* BLOCKING \*/

/\* **read and wait are all blocking**\*/

buffer[bytes\_read] = '\0';

printf( "Read %d bytes: %s\n", bytes\_read, buffer );

bytes\_read = read( p[0], buffer, 6 ); /\* BLOCKING \*/

buffer[bytes\_read] = '\0';

printf( "Read %d bytes: %s\n", bytes\_read, buffer );

int pid = fork(); /\* this will also copy fd table to child \*/

/\* fd tables (after parent closes p[1] and child closes p[0]):

PARENT CHILD

0: stdin stdin

1: stdout stdout

2: stderr stderr

3: p[0] <=======read

4: write========< p[1]

\*/

if ( pid == 0 ) /\* child \*/

{

close( p[0] ); /\* ensure no reads are done in the child \*/

int bytes\_written = write( p[1], "ABCDEFGHIJKLMNOPQRS", 10 );

/\* -- context switch could occur here, causing PARENT

output line to appear before the CHILD output line -- \*/

printf( "CHILD: Wrote %d bytes\n", bytes\_written );

}

else /\* parent \*/

{

close( p[1] ); /\* ensure no writes are done in the parent \*/

char buffer[10];

int bytes\_read = read( p[0], buffer, 6 ); /\* BLOCKING \*/

/\* -Ma

Blocking means waiting until this line is excuted

NoBlocking means don't wait

\*/

buffer[bytes\_read] = '\0';

printf( "PARENT: Read %d bytes: %s\n", bytes\_read, buffer );

}

int pid = fork(); /\* this will also copy fd table to child \*/

/\* fd tables (after parent closes p[1] and child closes p[0]):

...and after dup2() call

PARENT CHILD

0: stdin stdin

1: stdout write========< p[1]

2: stderr stderr

3: p[0] <=======read

4: write========< p[1]

\*/

if ( pid == 0 ) /\* child \*/

{

close( p[0] ); /\* ensure no reads are done in the child \*/

int bytes\_written = write( p[1], "ABCDEFGHIJKLMNOPQRS", 10 );

/\* -- context switch could occur here, causing PARENT

output line to appear before the CHILD output line -- \*/

printf( "CHILD: Wrote %d bytes\n", bytes\_written );

int rc = dup2( p[1], 1 );

/\* closes fd 1; redirect stdout (fd 1) to p[1] \*/

printf( "this should go to the pipe now...\n" );

}

else /\* parent \*/

{

close( p[1] ); /\* ensure no writes are done in the parent \*/

char buffer[100];

int bytes\_read = read( p[0], buffer, 100 ); /\* BLOCKING \*/

buffer[bytes\_read] = '\0';

printf( "PARENT: Read %d bytes: %s\n", bytes\_read, buffer );

bytes\_read = read( p[0], buffer, 100 ); /\* BLOCKING \*/

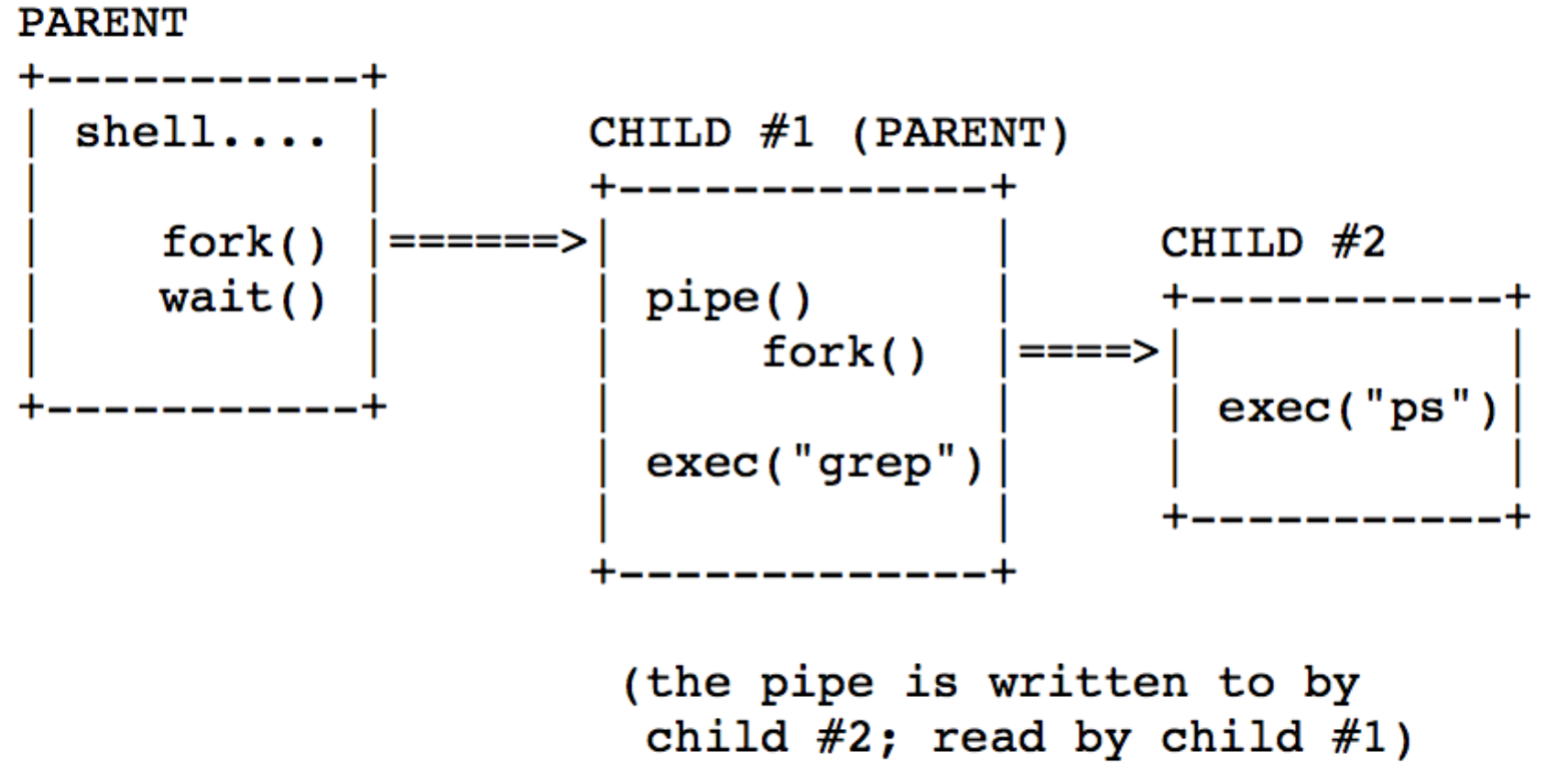
buffer[bytes\_read] = '\0';

printf( "PARENT: Read %d bytes: %s\n", bytes\_read, buffer );

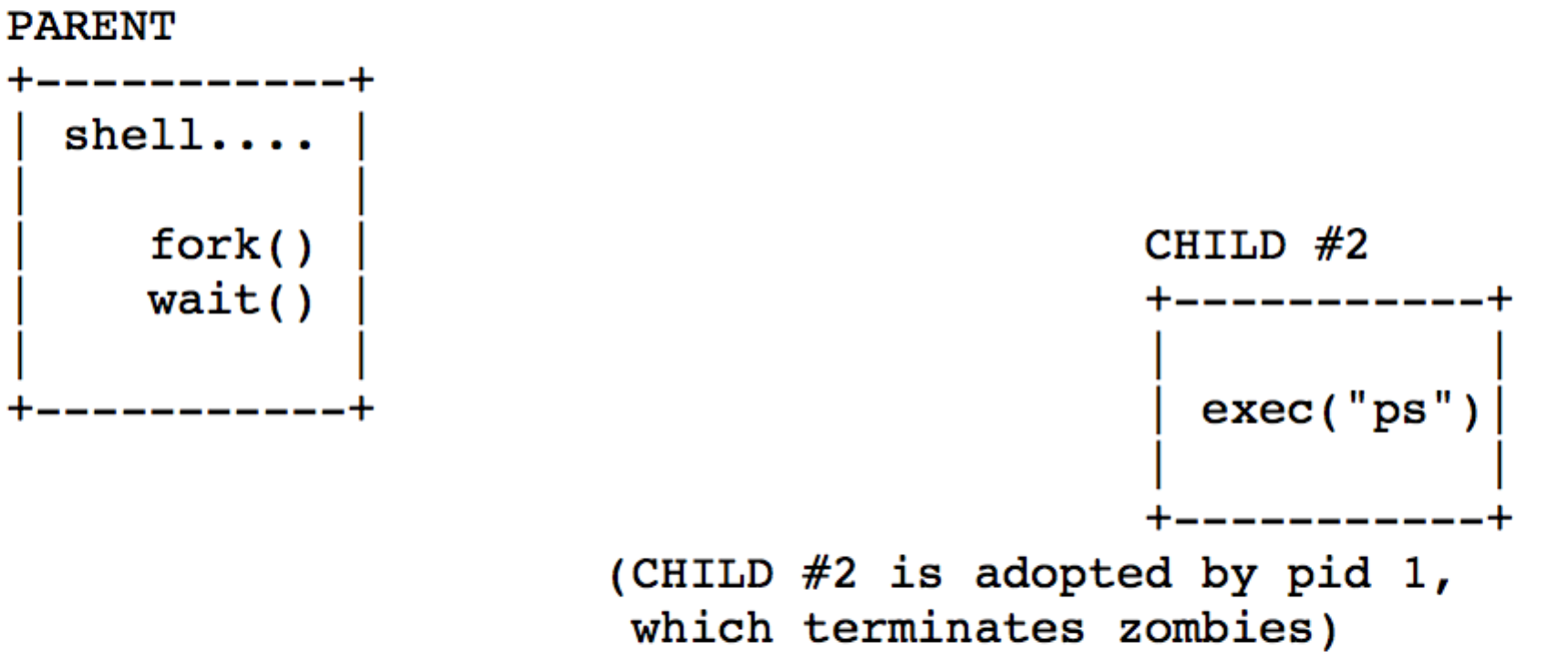
}

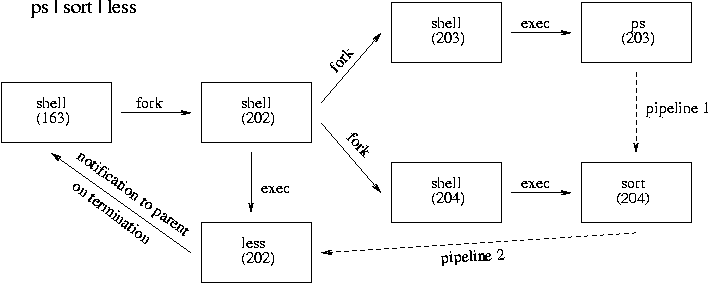
For the pipe functionality:

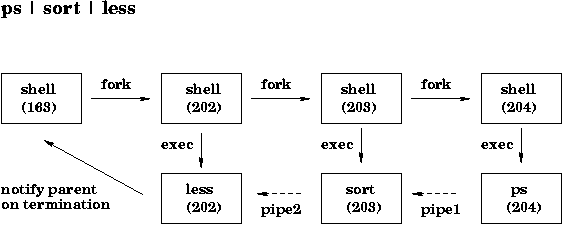
"ps -ef | grep cutler"



...after grep and ps processes terminate:







CPU SCHEDULING (a.k.a. SHORT-TERM SCHEDULING)

-- long-term scheduling: algorithm to select next job to be admitted to the system

-- job (a batch job waiting to enter the system)

-- process (a program in execution already in system)

Multiprogramming -- multiple processes residing in memory at once (requires CONTEXT SWITCHING)

Processes both COMPETE for resources (multiple process are in READY state) and often must COOPERATE with on another (IPC, Synchronization)

Given: n processes in READY state (in memory)

Assume: 1 CPU available

Algorithm: select next process to execute with CPU

does it matter (since computers are sooooo fast)?

-- on laptops/PCs, the "in-focus" app is the best candidate to execute next (e.g. text editor)

-- on (busy) servers, it matters....

why does it matter?

-- to support multiprogramming, we need context switches

(which is costly overhead, including switch from user mode to kernel mode)

-- aim to make most efficient use of CPU

-- aim to achieve FAIRNESS amongst all processes, though also support process priorities

**CPU-bound and I/O-bound processes**

-- Processes use the CPU in "bursts"

-- CPU runs a process until the process performs a system call requesting I/O

-- CPU-bound processes spend most of their time using CPU -- **longer CPU burst times**

-- I/O-bound processes spend most of their time waiting for I/O -- **shorter CPU burst times**

-- as CPUs get even faster, processes tend to be more I/O-bound, because CPUs are faster than disk, etc.

Preemption vs Nonpreemption

-- A nonpreemptive scheduling algorithm selects a process and allows it to run until the process "decides" to relinquish control of CPU (i.e. after it's executed its CPU burst)

-- A preemptive scheduling algorithm selects a process and allows it only to run at most some fixed timeslice (based on clock interrupt) or will preempt it (kick it out of the CPU) for some other reason

Process mix:

-- batch or cron processes (running in background)

-- interactive processes (requires preemption)

-- real-time processes (often designed to have shorter CPU bursts)

Given: n processes in READY state (in memory)

Assume: 1 CPU available

Algorithm: select next process to execute with CPU

given: n jobs with required CPU times

-- first job finishes at time A

-- second job finishes at time A+B

-- third job finishes at time A+B+C

-- etc.

(4A + 3B + 2C + D)

-- mean turnaround time = ------------------

4

-- intuitively, A contributes much more to the mean than D

-- disadvantages: assumes all jobs available at time 0

-- disadvantages: cannot typically know runtimes ahead of time

-- how to choose length of timeslice?

select timeslice to be much more than context switch time (e.g. 100ms vs 1ms gives 1% overhead)

e.g. 100ms timeslice; 50 processes in READY state; last process could wait 50x100ms=5000ms=5sec for CPU

if timeslice is longer than mean CPU burst, preemption will not happen all that often; instead, allow process to relinquish control of CPU this is better, because context switches occur only when logically necessary

if timeslice too short, too many context switches and therefore reduced CPU efficiency

if timeslice too long, poor response times to short CPU burst requests

Priority Scheduling

-- use round-robin for each priority level

-- problem: starvation of lower-priority processes(solution: aging; increase priority automatically as process ages in ready queues)

-- preemptive? if so, when a higher-priority process arrives, preempt the running lower-priority process

Scheduling algorithms:

e.g. CPU burst time

P1 13ms

P2 3ms

P3 5ms

-- assume all processes have arrived at time 0

-- ignore time required to add to the queue

-- ignore context switch time

FCFS (**First-come, first-served**)

-- nonpreemptive

-- simplest to understand, implement (FIFO queue), etc.

-- not very efficient

-- certainly fair

+-------------+---+-----+

| P1 |P2 | P3 |

+-------------+---+-----+

1111 111 11122

1234567890123 456 78901

CPU burst turnaround time wait time

(b) (t) (w)

P1 13ms 13ms 0ms

P2 3ms 16ms 13ms

P3 5ms 21ms 16ms

t = w + b

w = t - b

avg 16.67ms 9.667ms

SJF **(Shortest Job First)**

-- nonpreemptive

-- assumes CPU burst times are known in advance

-- key disadvantage: cannot know CPU burst times ahead of time

-- disadvantage: potential starvation for long-running processes

(e.g. P1) if shorter-running processes keep arriving

P1 13ms

P2 3ms

P3 5ms

+---+-----+-------------+

|P2 | P3 | P1 |

+---+-----+-------------+

111111111122

123 45678 9012345678901

CPU burst turnaround time wait time

(b) (t) (w)

P1 13ms 21ms 8ms

P2 3ms 3ms 0ms

P3 5ms 8ms 3ms

avg 10.67ms 3.667ms

Newly arriving processes (i.e. arrival time > 0)

CPU burst arrival time

P1 13ms 0ms

P2 3ms 0ms

P3 5ms 0ms

P4 3ms 4ms

+---+-----+---+-------------+

|P2 | P3 |P4 | P1 |

+---+-----+---+-------------+

11 1111111122222

123 45678 901 2345678901234

CPU burst turnaround time wait time

(b) (t) (w)

P1 13ms 24ms 11ms

P2 3ms 3ms 0ms

P3 5ms 8ms 3ms

P4 3ms 7ms 4ms

avg 10.5ms 4.5ms

Shortest Remaining Time (**SRT**)

Shortest Remaining Time Next (**SRTN**)

Shortest Process Next (**SPN**)

-- preemptive SJF

-- when a new job arrives, its required time is compared to remaining time of running process; if new job requires less time, preemption occurs

CPU burst arrival time

P1 13ms 0ms

P2 3ms 0ms

P3 5ms 0ms

P4 3ms 4ms

preemption

|

v

+---+-+---+----+-------------+

|P2 |3|P4 | P3 | P1 |

+---+-+---+----+-------------+

11 1111111122222

123 4 567 8901 2345678901234

-- is this better?!

-- may cause starvation in P1

(processes with longer CPU burst times)

-- additional context switches

-- P4 turnaround and wait times were reduced

(favor processes with shorter CPU burst times)

How do we know (guess) which currently runnable process is shortest (has shortest burst time)?

-- predict based on history

-- use a technique called exponential averaging

for \*each\* process

t is actual length of the nth CPU burst of P1

n

tau is predicted value of the nth CPU burst of P1

n

the estimate of next CPU burst of P1 is as follows:

tau = a t + (1-a) tau (with 0 < a < 1)

n+1 n n

TO DO: what does it mean when a = 0 or a = 1? use a = 0.5?

e.g. tau = 10 t = 6

0 0

tau = 0.5 t + 0.5 tau = 0.5 x 6 + 0.5 x 10 = 8

1 0 0

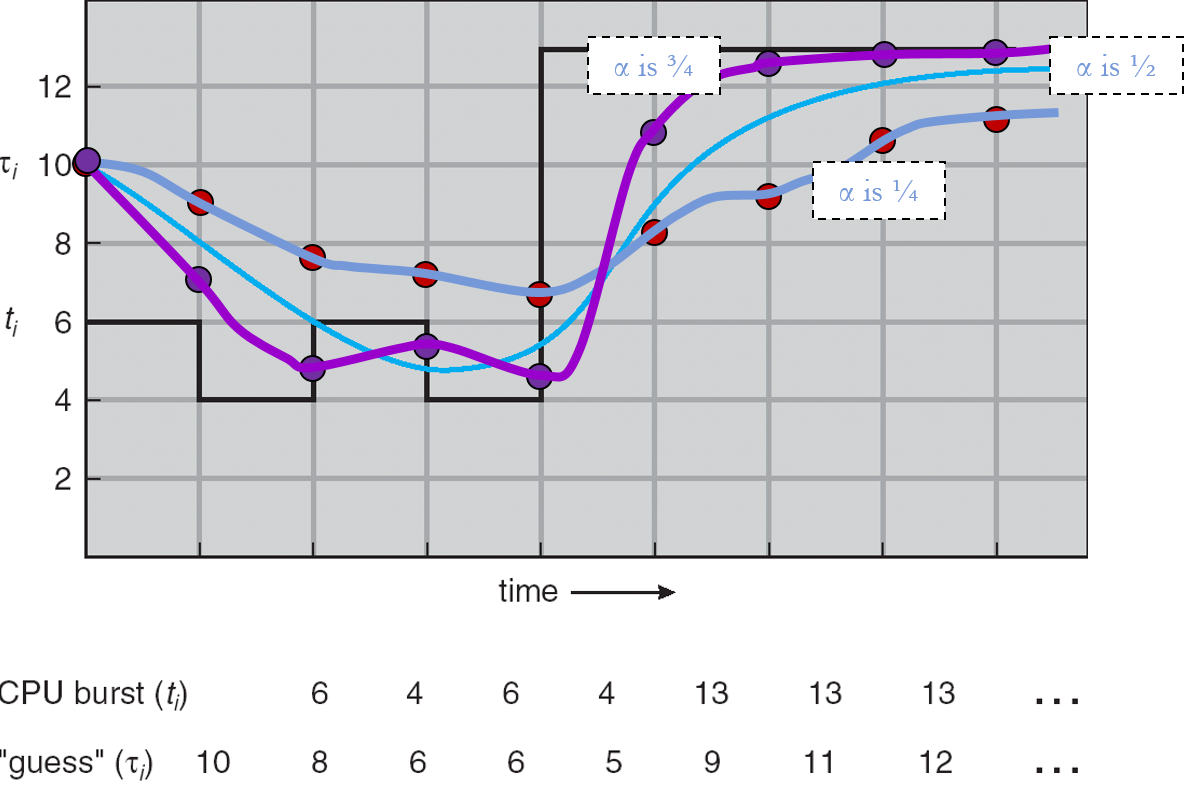
t = 4

1

tau = 0.5 t + 0.5 tau = 0.5 x 8 + 0.5 \* 4 = 6

2 1 1

etc.



**Round-Robin (RR)**

-- preemptive (based on fixed timeslice or quantum)

-- preemption occurs when timeslice expires

-- processes might opt to relinquish control of CPU sooner

-- simple, fair, most widely used

-- requires the use of a queue data structure

-- newly arriving processes go to the end of the queue

(alternatively, should we add them to the front of the queue to improve initial response times?)

CPU burst arrival time

P1 13ms 0ms

P2 3ms 0ms

P3 5ms 0ms

P4 3ms 4ms

assume timeslice is 2ms

INITIAL QUEUE: P1 P2 P3 <end>

QUEUE:

at time 4ms, P2 is returning to the end of the Q, and P4 arrives; either Q order is fine here (negligible) (assume P4 added before P2)

|

P1 >XXp XXp XXp XXXXXXX.

P2 > XXp X.

P3 > XXp XXp X.

P4 | > XXp X.

+---------------------------------->

111111111122222

123456789012345678901234

CPU burst turnaround time wait time

(b) (t) (w)

P1 13ms 24ms 11ms

P2 3ms 11ms 8ms

P3 5ms 17ms 12ms

P4 3ms 12ms 9ms

avg 16ms 10ms

-- is this the worst algorithm?!

-- if so, why is it used in practice?

-- no starvation

-- fairness

-- short response times for short-bursted processes

(interactive processes)

-- often used as part of a multilevel queue system

Selecting an optimal time slice value

t (slice) must be large as compared to t (cs) (otherwise, overhead too high.....)

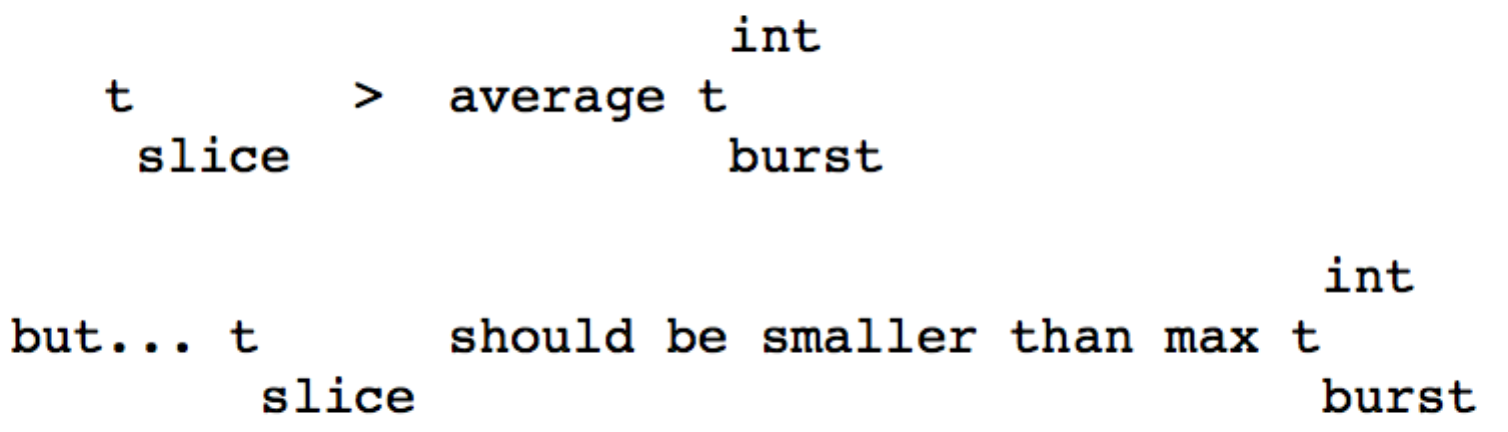
note that t (cs) includes saving/loading PCBs and the dispatcher's scheduling decision

large t (slice) values lead to FCFS/FIFO

small t (slice) values lead to scenario in which

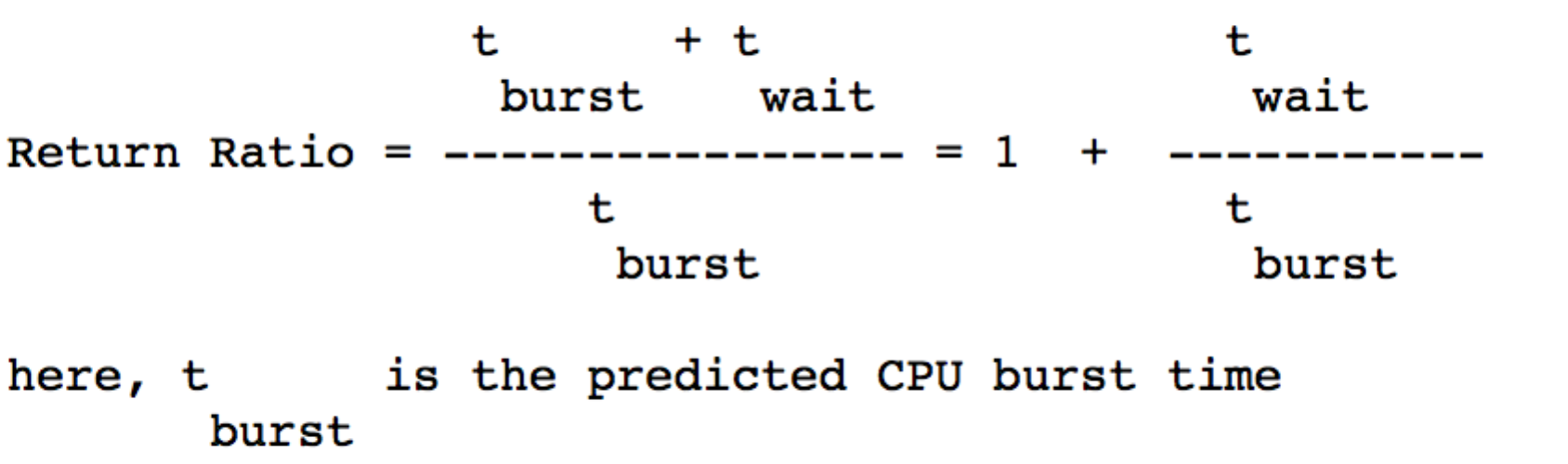
each process appears to run on its own dedicated processor whose speed is 1/n of the speed of the actual processor, where n is the number of processes in the ready queue

goal: majority of interactive processes should reach their "organic" burst time, i.e. relinquish the CPU because they want to do I/O (or terminated)



Highest return ratio next (**HRRN**)

-- the higher the return ratio, the higher the process priority



-- disadvantages: starvation

(low priority processes may never execute, but aging might help)

Multilevel Queues (MQs)

-- a multi-class system is one in which there are several classes of processes

-- each class of process may be assigned a different priority, serviced via MQs

-- each class potentially uses a different scheduling algorithm

e.g. interactive vs. batch processes

interactive/foreground processes use RR

batch/background processes use FCFS

-- need a mechanism for scheduling between queues (e.g. RR with preemption?)

MQ Fixed Priority Scheduling

-- Multiple queues A, B, C, ....

-- Serve all processes from A, then from B, ....

-- If servicing a lower-level queue and a process arrives on a higher-level queue, start servicing the higher-level queue again either:

-- preemptively (i.e. preempt current running lower-level process)

-- nonpreemptively (wait until current running lower-level process

finishes its burst)

-- Starvation

MQ Time Quantum (Slice) Allocation

-- Multiple queues A, B, C, ....

-- Each queue is assigned a percentage of CPU time with which it can schedule amongst its processes

-- e.g. 80% of CPU time to queue A (RR)

20% of CPU time to queue B (FCFS)

-- Starvation still possible, since processes do not between queues

Multilevel Feedback Queues (MFQs)

-- Multiple queues A, B, C, ....

-- Processes can move between queues

-- Dynamically assign processes to multiple queues based on actual CPU burst times (i.e. feedback)

e.g.

-- New process P enters queue A (RR w t (slice) = 8ms)

-- If P not finished after 8ms burst, it goes to queue B (RR w t (slice) = 16ms)

-- If P not finished after 16ms burst (24ms thus far), it goes to queue C (FCFS)

-- For multiple queues, preemption is used (to avoid processes in queues B and C (especially C) from using "too much" CPU time, thus delaying processes in queue A (or B)

Many configuration options:

-- Number of queues / levels

-- Scheduling algorithm (and corresponding parameters) for each queue

-- Criteria for moving down (or up) from current queue

-- Which queue does the process enter first?

Multiple Processors

-- simplest approach is to use a single ready queue and have the dispatcher (or other mechanism) distribute the work to CPUs (doesn't change scheduling algorithm) (assumes processors are tightly coupled, i.e. they share memory and other resources)

