**False:**

- In RR, best performance is when time quantum is small w/ respect to CS time.

- In Linux Completely Fair Scheduler processes w/ low nice values always run before those w/ high nice values

- Thrashing is result of insufficient multiprogramming

- If a page’s contents in memory are different than they are on disk, the valid bit must be set to invalid

- illegal memory access results in page fault

- translation lookaside buffer (TLB) is used to search for free space in physical memory

void sched (void) {

struct proc \*p = myproc();

swtch(&p->context, &mycpu()->context); }

sched() calls swtch() to perform a context switch from the context of the kernel code that interrupted the currently executing process into the context of the scheduler. The purpose is for the scheduler to possibly context switch to a different process. swtch() performs the actions of a context switch, saving the current CPU registers to the process’ context and loading CPU registers from the saved context.

**ready->running**: there is a context switch. The scheduler replaces the currently running process and replaces it with the next process from the ready queue.

**running->blocked**: there is a context switch. The currently running process is placed into a blocked state, the scheduler replaces it with the next process from the ready queue

**blocked->ready**: no context switch. When a process is ready to run its placed in the ready queue, assuming there is no preemption, the currently running process remains running

*What are addresses using binary SI units for* 5287601 (\_/2^20 = 5MB). 17573467 (\_/2^20 = 17MB). 8731878194284 (\_/2^40 = 8TB)

*How a process’ physical memory can be relocated using base and bounds*: 1) if process currently running cause a trap to switch out of the process. 2) copy the contiguous block of physical memory to another location. 3) update base register to be start address of the new location in physical memory. Static relocation of physical process memory requires identifying all pointers in the executing program and updating them to point to the new location in physical memory. Without making assumptions about the program doing this is difficult to do.

*What is the size of addressable memory for* 16-bit address (2^16 bytes). 24-bit address (2^24 bytes). 32-bit address (2^32 bytes)

**fork()**: makes copy of the current process

**CPU bound**: process that frequently exceeds its time slice

**Semaphore**: mechanism that provides both locking and signaling

**TCB**: created from pthread\_create() call

**Kernel mode**: system calls are included using traps b/c its requires this

**Pipe**: queue for inter-process communication

**Locality**: reason caching can improve performance

**TestAndSet**: machine instruction that can be used to implement mutex locks

**True:**

- LRU never experiences Belady’s Anomaly

- purpose of priority boost is to prevent starvation

- segment can be shared by multiple processes

- paging has the problem of internal fragmentation

- bounded buffer problem can be solved using only semaphores to control concurrency

- binary semaphore is equivalent to a mutex lock

- thread requesting a resource never causes a deadlock if that thread doesn’t currently have any other resources assigned to it

**System call**: call a system library and transfer control to the OS in kernel mode

**Interrupt**: signal from hardware source that gives CPU control to the interrupt handler

**Trap**: signal from a software source that gives CPU control to the interrupt handler

System calls need to execute in kernel mode, kernel mode can be entered by an interrupt or trap. Software initiates a system call by causing a trap. Hardware changes control to the OS by an interrupt.

*Virtual address space of 1024 pages, 4096 byte page size, how many bits required to store virtual address?* 1024+4096 = 2^10 + 2^12 = 22

**fork bomb**:

int main(int argos, char \*argv[]) { int rc=0; while(rc == 0) { rc=fork(); } wait(NULL); return 0; }

The result of the loop is the child forks a new process and this happens recursively. So an infinite number of processes are created. The parents call wait so they never terminate. Eventually the system will run out of memory or process ids. To prevent, set a max number of processes.

**Race condition** for void bid(double amount) { if(amount > highestBid) { highestBid = amount; } }. The easiest way to fix is use a mutex lock.

void bid(double amount { acquire(mutex); if(amount > highestBid) { highestBid = amount; } release (mutex); }

**FIFO**: starvation. Job can have infinite loop and starve other jobs

**STCF**: starvation. Many short jobs prevent longer jobs having runtime

**RR**: no starvation. Process execution within (N\* time slice)

**Lottery scheduling**: no starvation. Every lottery selection process has chance of executing based on how many tickets it has

I/O bound is voluntary (more likely to wait for I/O to become available before using up its time slice). It’s more likely to voluntarily relinquish the CPU. CPU bound is nonvoluntary. The program’s likely to use the CPU for entire time quantum.

sem\_t customersReadyToOrder = 0; sem\_t chefTakingOrder;

sem\_t customerGivingOrder; sem\_t chefServingFood;

sem\_t seatsMutex = 0;  int seatsAvailable = N;

void init() {

sem\_init(customersReadyToOrder, shared, 0);

sem\_init(chefTakingOrder, shared, 0);

sem\_init(customerGivingOrder, shared, 0);

sem\_init(chefServingFood, shared, 0);

sem\_init(seatsMutex, shared, 0);  int seatsAvailable = N; }

void chef() {

while (true) {

sem\_wait(customersReadyToOrder); printf("Welcome to the sushi bar.\n");

sem\_post(chefTakingOrder); printf("What will you have?\n");

sem\_wait(customerGivingOrder); printf("Here is your food.\n");

sem\_post(chefServingFood);  } }

void customer() {

sem\_wait(seatsMutex);

if (seatsAvailable == 0) {

sem\_post(seatsMutex); printf(“I’m not waiting around, bye.\n”);

return; }

printf(“This seat looks good.\n”);

seatsAvailable--; sem\_post(seatsMutex);

sem\_post(customersReadyToOrder); sem\_wait(chefTakingOrder);

printf("Hello.\n"); printf("I will have one roll.\n");

sem\_post(customerGivingOrder); sem\_wait(chefServingFood);

printf("That was good.\n");  set\_wait(seatsMutex);

seatsAvailable++;  sem\_post(seatsMutex); }

Consider system w/ virtual address space 4096 pages, 2KB page size. Physical memory w/ max capacity of 1024 frames.

How many bits for addresses in virtual address space? 4096\*2048 = 2^23 bytes. 23 bits

Bits for addresses in physical memory? 1024\*2048 = 2^21 bytes. 21 bits

RR has timer preemptions that cause extra context switches that STCF won’t have. Context switches contribute to significant overhead of the scheduler. RR is less CPU efficient.

*Suppose a machine has 38 bit virtual addresses and 32 bit physical addresses.*

Main advantage of multilevel page table: doesn’t require page table entries for every unallocated page.

*What is required to support dynamic memory allocation?*

**Base and bounds**: copy process to new location in physical memory. Then update the base and bounds registers

**Segmentation**: if there’s free space after the segment then the segment can be expanded by updating the size register for that segment. If there isn’t enough free space after the segment then it must be copied to a new physical location and the base and bounds registers updated

**Paging**: allocated free physical frames to unused pages by updating the page table

virtual\_address+base=physical\_address

**Base and bounds**: External fragmentation (yes). Internal fragmentation (no). Variable sized allocations can lead to free spaces that aren’t large enough to fit an entire process.

**Segmentation**: External fragmentation (yes). Internal fragmentation (no). Variable sized allocations can lead to free spaces that aren’t large enough to fit an entire segment

**Paging**: External fragmentation (no). Internal fragmentation (yes). Pages and frames are the same size so there will never be wasted external free memory that can’t be allocated to a page. Its possible for the allocated memory to not completely fill a page, so

there can be internal fragmentation.

**Race condition**: variable num\_of\_proc. A call to acquire() must be placed upon entering each function and a call to release() immediately before exiting each function.

#define MAX\_PROCESSES 255

int num\_of\_proc = 0;

int allocate\_process () { int new\_pid; if (num\_of\_proc == MAX\_PROCESSES) { return -1; } else { num\_of\_proc++; return new\_pid; } }

void release\_process () { num\_of\_proc–; }

#include <pthread.h>

sem\_t ping; sem\_t pong;

void init() {

sem\_init(&ping, share, 0);

sem\_init(&pong, share, 0); }

void\* ping\_thread(void \*arg) {

while(1) { printf(“ping\n”); sem\_post(&pong); sem\_post(&ping); } }

void\* pong\_thread(void \*arg) {

while(1) { sem\_wait(pong); printf(“pong\n”); sem\_post(&ping); } }

22000 -> 4KB is 4096. 22000/4096 = 5. 22000%4096 = 1520.

77056 -> 77056/4096 = 18. 77056%4096 = 3328.

197012 -> 197012/4096 = 48. 197012%4096 = 404.

Given the reference string of page accesses: 4 3 2 1 4 3 5 1 and a system with 3 page frames, *how many page faults result when using the following page replacement policies?*

(a) FIFO # page faults = 8

A grid of numbers and letters

Description automatically generated

(b) OPT # page faults = 5

A grid of numbers and letters

Description automatically generated

Implement methods using only **locks and condition variables to manage concurrency.**

cond\_t cond; mutex\_m mutex; int available;

void init (int n) { available = n; }

void request (int n) {

pthread\_mutex\_lock(&mutex);

while (n > available) {

pthread\_cond\_wait(&cond, &mutex); }

available -= n;

pthread\_mutex\_unlock(&mutex); }

void release (int n) {

pthread\_mutex\_lock(&mutex);

available += n; pthread\_cond\_signal(&cond);

pthread\_mutex\_unlock(&mutex); }

Process’ logical address consists of 4 pages with size 2KB. Compute the physical address for the logical addresses.

(1018) Page # = 1018/2048 = 0. Offset = 1018. Page 0 maps to frame 1. Frame 1 starts at address 2048.

Physical address = 1018+2048 = 3066

(6976) Page # = 6976/2048 = 3. Offset = 6976%2048 = 832. Page 3 maps to frame 7. Frame 7 starts at address 7\*2048 = 14336. Physical address = 14336+832 = 15168

Afgdfg

3 programs are serviced in a multiprogramming system. Program A contains 50ms of computation followed by 100ms of I/O on hardware device 1. B contains 20ms of computation followed by 50ms of I/O on hardware device 2. C contains 50ms of computation followed by 100ms I/O on hardware device 2. Each device can service only 1 I/O request at a time. What is the minimum time it will take to complete all 3 programs? Create a table to show the operation of the three programs.

A white table with black text

Description automatically generated