T-106.4155 Operating systems

0 (1p) Fill in the feedback form available in the Course Noppa News page.

1 (10p) Give short definitions for the following. (One point per question.)

a) What is double buffering?

(Stallings pp. 506) In double buffering, a process transfers data to (or from) one buffer while the operating system empties (or fills) the other.

b) What are continous and chained file allocation methods?

(Stallings pp. 572–573) In contiguous allocation a single contigous set of blocks is allocated to a file at the fime of file creation. In chained allocation, each block has a pointer to the next block in the chain.

c) What is a process?

(Stallings pp. 790) A program in execution. A process is controlled by the operating system. Same as *task*

d) What is an interrupt?

(Stallings pp. 788) A suspension of a process, such as the execution of a computer program, cased by an event external to that process and performed in such a way that the process can be resumed.

e) What is an asynchronous operation?

(Stalllings pp. 785) An operation that occurs without a regular or predictable time relationship to a specified event.

f) What is DMA?

(Stallings pp. 786) A form of I/O in which a special module, called DMA module, controls the exchange of data between main memory and I/O device. The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

g) What is MMU?

Memory Management Unit translates virtual addresses into physical addresses, handles memory protection and cache control.

h) What is priority inversion?

(Stallings pp. 790) A circumstance in which the operating system forces a higehr priority task to wait for a lower-priority task.

i) What is external fragmentation?

(Stallings pp. 787) Occurs when memory is divided into variable-size partitions corresponding to the blocks of data assigned to the main memory (e.g. segemnts in main memory). As segments are moved into and out of the memory, gaps will occur between the occupied portions of memory.

j) What is a privileged instruction?

(Stallings pp. 790) An instruction that can be executed only in a specified mode, usually by a supervisory program.

Note that long explanations (several sentences) are *not allowed*.

2 (6p) Consider the producer-consumer problem: one process reading from and one process writing into an N element *shared buffer*. Give a solution that implements mutual exclusion by using a *monitor*.

You can use either *Hoare* or *Lampson–Redell* monitors but indicate which one you are using. Present your solution as a piece of pseudo code and give a short explanation. Provide only the monitor definition, i.e. `append()`, `take()` methods and other internal definitions for the monitor.

Stallings pp. 227-229

Hoare's monitor:

```
monitor producerconsumer
char buffer [N];
int nextin, nextout;
int count;

void append(char x)
{
  if (count == N) cwait(notfull);
  buffer[nextin] = x;
  nextin = (nextin + 1) % N;
  count++;
  csignal(notempty);
}

void take(char x)
{
  if (count == 0) cwait(notempty);
  x = buffer[nextout];
  nexout = (nextout + 1) % N;
  count--;
```

```
  csignal(notfull);
}

{ nextin = 0 ; nextout = 0; count = 0; }
```

Lampson–Redell monitor:

```
monitor producerconsumer
char buffer [N];
int nextin, nextout;
int count;

void append(char x)
{
  while (count == N) cwait(notfull);
  buffer[nextin] = x;
  nextin = (nextin + 1) % N;
  count++;
  cnotify(notempty);
}

void take(char x)
{
  while (count == 0) cwait(notempty);
  x = buffer[nextout];
  nexout = (nextout + 1) % N;
  count--;
  cnotify(notfull);
}

{ nextin = 0 ; nextout = 0; count = 0; }
```

Main program

```
void producer()
{
  char x;
  while (true) {
    produce(x);
    append(x);
  }
}

void consumer()
{
  char x;
```

```
  while (true) {
    take(x);
    consume(x);
  }
}

void main()
{
  parbegin(producer, consumer);
}
```

Grading (max 6p)

- solution doesn't deal with the producer-consumer problem => 0p
- correct solution => 4p
- explaining Hoare vs. Lampson-Redell => 1p
- explanation of the code => 1p

Mistakes

- solution is not monitor => 0p
- acquire but no signaling => 4p
- acquire but no waiting => 2p

3 (4p) How is the address translation from virtual addresses to physical addresses done in a modern operating system? What kind of hardware is available to support such translations in modern systems?

Grading (max 4p)

- Overall picture of the page/frame structure (1p)
- MSB of the address is translated with the page table (1p)
- Detailed description of the page table (2-/3-level, IPT) (1p)
- TLB and MMU hardware (1p)

4 (4p) Assume that we have a disk with 200 tracks and a disk scheduler receiving track requests 55,58,39,18,90,160,150,38,184 in that order. Give a list of tracks accessed in the access time order and the average seek time of the sequence when scheduling policy is a) First-in-First-Out (FIFO) b) Shortest-Service-Time-First (SSTF) c) SCAN d) C-SCAN.

Assume that the disk head is initially located at track 100.

From Stallings pp. 512

```
REQUESTS 55,58,39,18,90,160,150,38,184  SUM  AVG
-----------------------------------------------
FIFO     55,58,39,18,90,160,150,38,184  497 55.3
SSTF     90,58,55,39,38,18,150,160,184  248 27.5
SCAN     150,160,184,90,58,55,39,38,18  250 27.8
C-SCAN   150,160,184,18,38,39,55,58,90  322 35.8
```

SCAN and C-SCAN to decreasing track numbers:

```
SCAN     90,58,55,39,38,18,150,160,184   248 27.5 (same as SSTF)
C-SCAN   90,58,55,39,38,18,184,160,150   282 31.3
```

Grading

- 1p for each algorithm (0,5p access, 0,5p average or sum)
- for SCAN and C-SCAN the student can decided whether the head is initally moving towards increasing or decreasing track number. (In the above, it is moving towards increasing track number.)
- the question asks for *average seek time* even though it is hard to calculate with the facts presented.
  - giving *average seek length* or some parameterized seek time given the seek length is ok

5 (6p) Write an one-page essay on kernel and user space threads.

(Stallings pp. 168–173)

- User-Level Threads (ULT)
  - all threads are executed in a single OS process; OS is totally unaware of user-level threads
  - thread implementation is usually in a userland library that contains necessary code for creating and mainting threads, doing the scheduling
- Kernel-Level Threads (KLT)

  - a process contains multiple kernel-level threads that are scheduled by the OS kernel
  - kernel level threads share the same *address space* but have their own execution stacks in the process control block
- ULT vs. KLT
  - ULT context switch is faster; no need to jump to kernel and invoke its scheduling routines
  - In ULT, it is easy to do application specific scheduling
  - ULTs can be run on any OS provided that sufficient functionality in the OS (e.g. `getcontext()` and `setcontext(2)`) is available
  - In ULT, if a thread invokes a *blocking system call*, the whole process is stopped until the call returns; KLT's only the thread executing blocking call is blocked. (This can be worked around by either having separate processes or *jacketing*, i.e. converting blocking system calls into non-blocking by first checking whether the I/O call would block before actually performing it.)
  - In KLT, threads can be executed truly in parallel, e.g. in SMP setup. in ULT all threads are confined into single process executed in a single CPU by the OS kernel.
- Combined Approaches
  - we can combine both ULTs and KLTs; single process can contain multiple parallely executable KLTs that each have multiple ULTs
  - we get all the parallelism provided by the hardware
  - system calls don't block the entire execution
  - Programmer can choose the number of ULTs and KLTs accordingly

Grading

- Explaining ULT => 1p
- Explaining KLT => 1p
- Combined approaches => 1p
- ULT vs. KLT Comparison => 3p